Picamerax 21.9.8 Documentation

Release 21.9.8

Dave Jones

Jun 23, 2022

Contents

CHAPTER 1

Installation

It is simplest to install system wide using Python's `pip` tool:

```
$ sudo pip install picamerax
```

If you wish to use the classes in the *picamerax.array* (page 107) module then specify the "array" option which will pull in numpy as a dependency:

```
$ sudo pip install "picamerax[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 picamerax
```

If you ever need to remove your installation:

```
$ sudo pip uninstall picamerax
```

1.1 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 picamerax library attempts to maintain backward compatibility with older Pi firmwares, it is only tested against the latest firmware at the time of release, and not all functionality may be available if you are running an older firmware. As an example, the `annotate_text` attribute relies on a recent firmware; older firmwares lacked the functionality.

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

```
$ uname -a
```

The firmware revision is the number after the #:

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

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

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

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

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

You will need to reboot after doing so.

Note: Please note that the PiTFT[1] 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.

[1] https://www.adafruit.com/product/1601

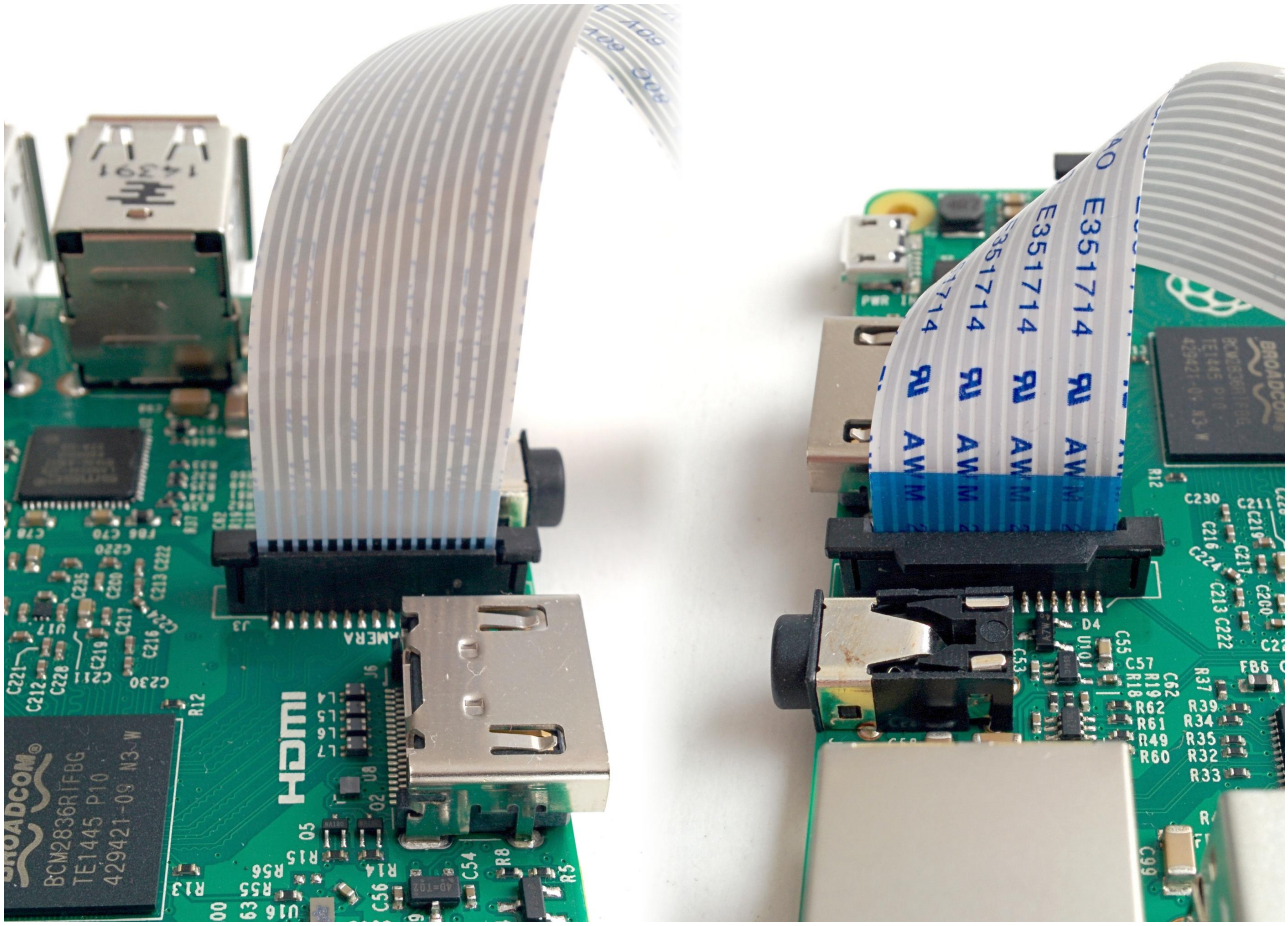
CHAPTER 2

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:

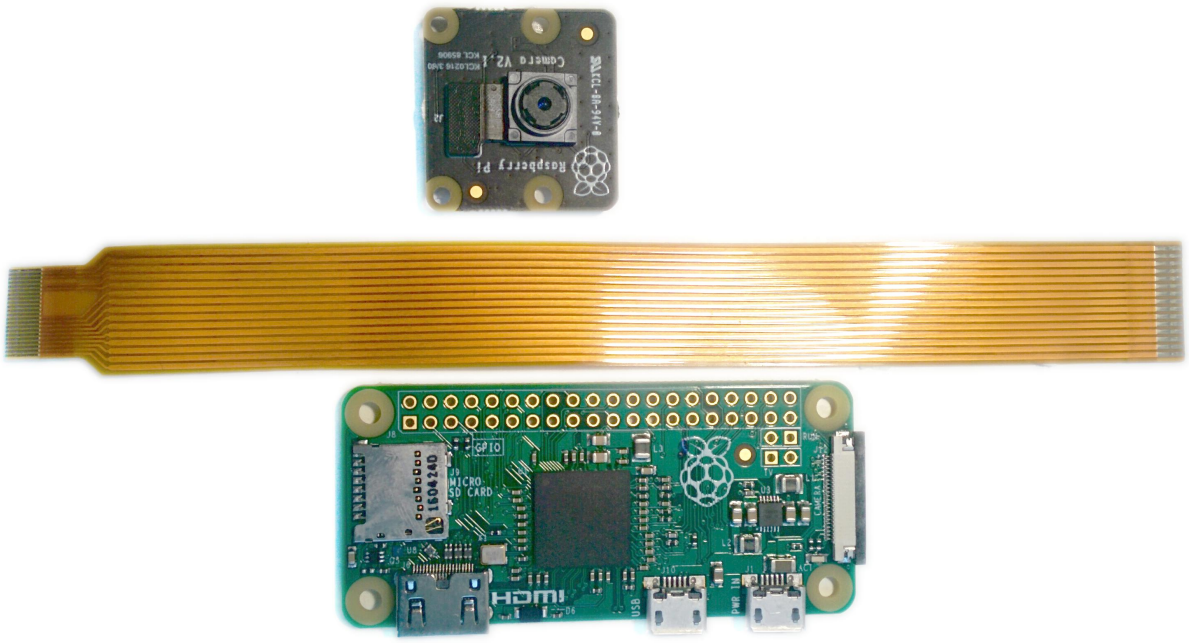


Make sure the camera module isn't sat on anything conductive (e.g. the Pi's USB ports or its GPIO pins).

2.1 Pi Zero

The 1.2 model of the Raspberry Pi Zero[2] includes a small form-factor CSI port which requires a camera adapter cable[3].

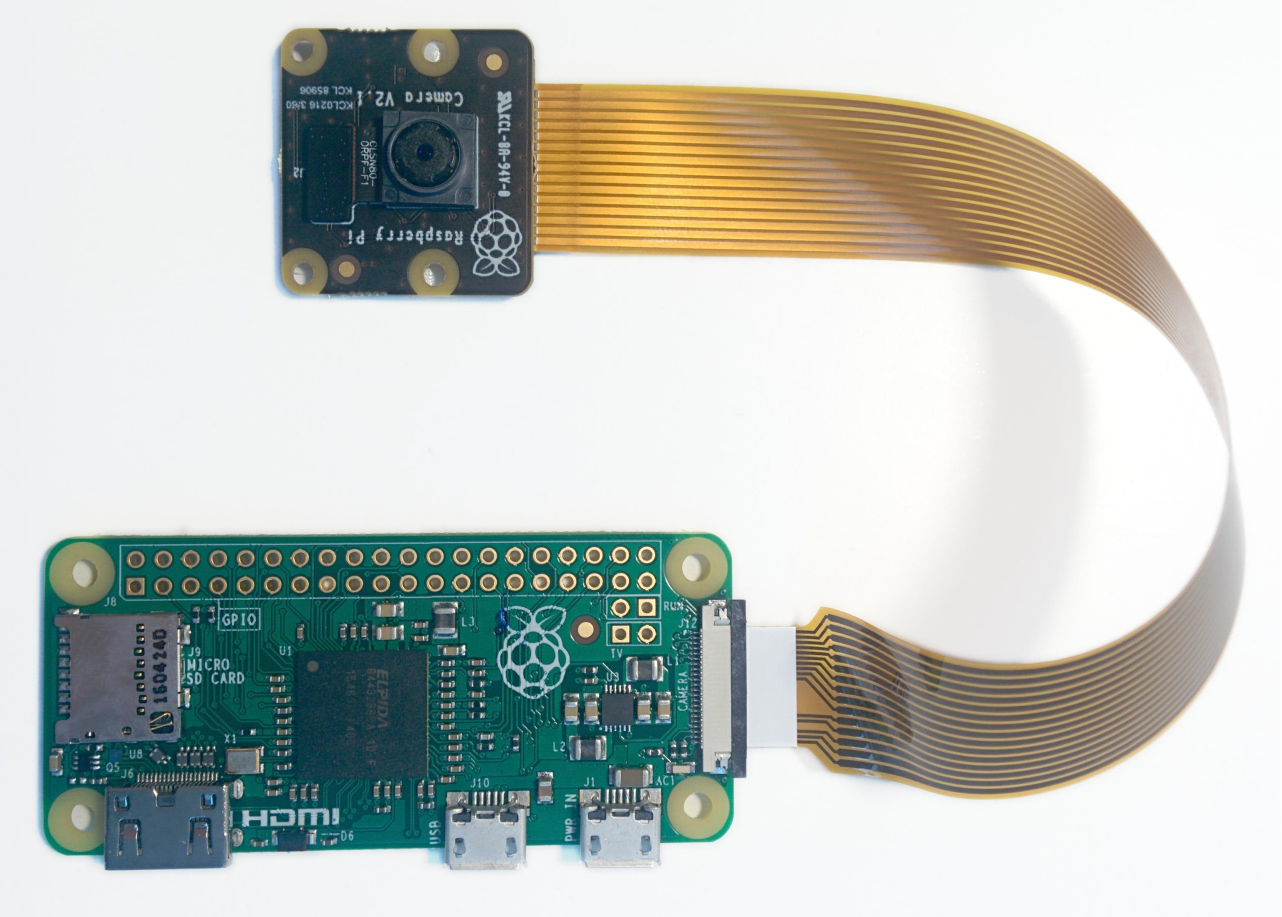
[2] https://www.raspberrypi.org/products/pi-zero/
[3] https://shop.pimoroni.com/products/camera-cable-raspberry-pi-zero-edition



To attach a camera module to a Pi Zero:

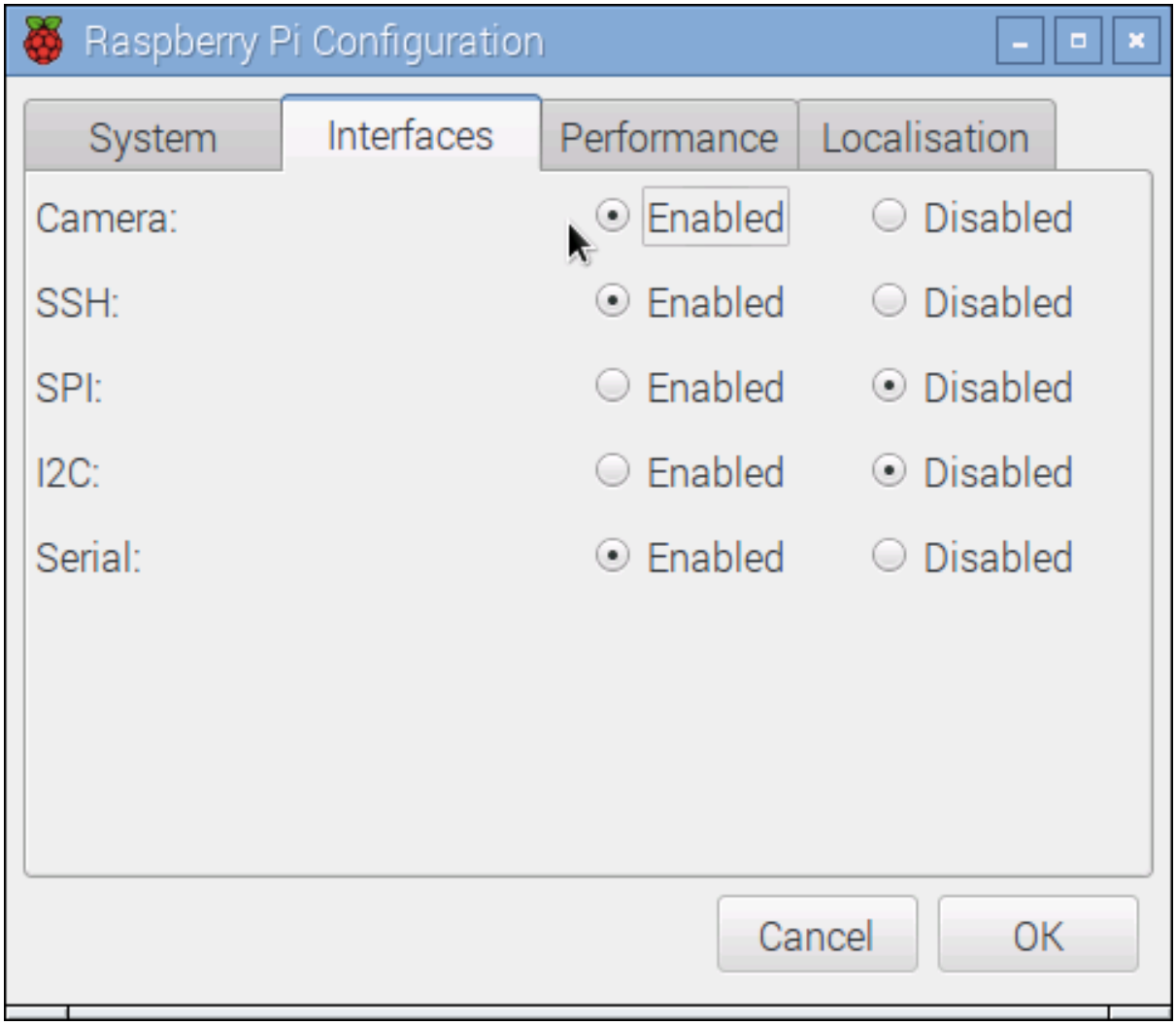
1. Remove the existing camera module's cable by gently lifting the collar on the camera module and pulling the cable out.
2. Next, insert the wider end of the adapter cable with the conductors facing in the same direction as the camera's lens.
3. Finally, attach the adapter to the Pi Zero by gently lifting 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:



2.2 Testing

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



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

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

If everything is working correctly, the camera should start, a preview from the camera should appear on the display and, after a 5 second delay it should capture an image (storing it as `image.jpg`) before shutting down the camera. Proceed to the *Basic Recipes* (page 9).

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 `picamerax.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).

3.1 Capturing to a file

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

```
from time import sleep
from picamerax 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 picamerax (as in the case above) will be flushed and closed so that when the `capture()` method returns, the data should be accessible to other processes.

3.2 Capturing to a stream

Capturing an image to a file-like object (a `socket()`[4], a `io.BytesIO`[5] stream, an existing open file object, etc.) is as simple as specifying that object as the output of whatever `capture()` method you're using:

[4] https://docs.python.org/3.5/library/socket.html#socket.socket
[5] https://docs.python.org/3.5/library/io.html#io.BytesIO

```
from io import BytesIO
from time import sleep
from picamerax 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`[6] 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; picamerax 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 picamerax 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`[7]) you can also construct your own *custom outputs* (page 29).

3.3 Capturing to a PIL Image

This is a variation on *Capturing to a stream* (page 9). First we'll capture an image to a `BytesIO`[8] 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[9] Image object:

```
from io import BytesIO
from time import sleep
from picamerax 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
```

(continues on next page)

[6] https://docs.python.org/3.5/library/io.html#io.BytesIO
[7] https://docs.python.org/3.5/library/io.html#io.BytesIO
[8] https://docs.python.org/3.5/library/io.html#io.BytesIO
[9] http://effbot.org/imagingbook/pil-index.htm

(continued from previous page)

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

3.4 Capturing resized images

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

```
from time import sleep
from picamerax import PiCamera

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

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

3.5 Capturing consistent images

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

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

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

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

```
from time import sleep
from picamerax 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
```

(continues on next page)

(continued from previous page)

```
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)])
```

3.6 Capturing timelapse sequences

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

```
from time import sleep
from picamerax 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`[10] 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 picamerax import PiCamera
from datetime import datetime, timedelta

def wait():
    # Calculate the delay to the start of the next hour
    next_hour = (datetime.now() + timedelta(hours=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()
```

3.7 Capturing in low light

Using similar tricks to those in *Capturing consistent images* (page 11), the Pi's camera can capture images in low light conditions. The primary objective is to set a high gain, and a long exposure time to allow the camera to gather as much light as possible. However, the `shutter_speed` attribute is constrained by the camera's `framerate` so the first thing we need to do is set a very slow framerate. The following script captures an image with a 6

[10] https://docs.python.org/3.5/library/datetime.html#module-datetime

second exposure time (the maximum the Pi's V1 camera module is capable of; the V2 camera module can manage 10 second exposures):

```
from picamerax 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[11]. 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` instead of `framerate`. 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.

3.8 Capturing to a network stream

This is a variation of *Capturing timelapse sequences* (page 12). 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[12] 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):

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

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

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

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

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

```
import io
import socket
import struct
import time
import picamerax

# 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 = picamerax.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()
```

(continues on next page)

(continued from previous page)

```
    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.

3.9 Recording video to a file

Recording a video to a file is simple:

```
import picamerax

camera = picamerax.PiCamera()
camera.resolution = (640, 480)
camera.start_recording('my_video.h264')
camera.wait_recording(60)
camera.stop_recording()
```

Note that we use `wait_recording()` in the example above instead of `time.sleep()`[13] which we've been using in the image capture recipes above. The `wait_recording()` method is similar in that it will pause for the number of seconds specified, but unlike `time.sleep()`[14] 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()`[15] instead, such errors would only be raised by the `stop_recording()` call (which could be long after the error actually occurred).

3.10 Recording video to a stream

This is very similar to *Recording video to a file* (page 15):

```
from io import BytesIO
from picamerax import PiCamera

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

(continues on next page)

[13] https://docs.python.org/3.5/library/time.html#time.sleep
[14] https://docs.python.org/3.5/library/time.html#time.sleep
[15] https://docs.python.org/3.5/library/time.html#time.sleep

(continued from previous page)

```
camera.wait_recording(15)
camera.stop_recording()
```

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

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

As well as using stream classes built into Python (like `BytesIO`[16]) you can also construct your own *custom outputs* (page 29). This is particularly useful for video recording, as discussed in the linked recipe.

3.11 Recording over multiple files

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

```
import picamerax

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

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

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

```
import picamerax

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

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

3.12 Recording to a circular stream

This is similar to *Recording video to a stream* (page 15) but uses a special kind of in-memory stream provided by the picamerax library. The `PiCameraCircularIO` class implements a ring buffer[17] 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

[16] https://docs.python.org/3.5/library/io.html#io.BytesIO
[17] https://en.wikipedia.org/wiki/Circular_buffer

write_now function returns True (in this implementation this is random but one could, for example, replace this with some sort of motion detection algorithm). Once write_now returns True, the script waits 10 more seconds (so that the buffer contains 10 seconds of video from before the event, and 10 seconds after) and writes the resulting video to disk before going back to waiting:

```
import random
import picamerax

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

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

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

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

New in version 1.0.

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

3.13 Recording to a network stream

This is similar to *Recording video to a stream* (page 15) but instead of an in-memory stream like BytesIO[18], we will use a file-like object created from a socket()[19]. Unlike the example in *Capturing to a network stream* (page 13) 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)
```

(continues on next page)

[18] https://docs.python.org/3.5/library/io.html#io.BytesIO
[19] https://docs.python.org/3.5/library/socket.html#socket.socket

(continued from previous page)

```
# Accept a single connection and make a file-like object out of it
connection = server_socket.accept()[0].makefile('rb')
try:
    # Run a viewer with an appropriate command line. Uncomment the mplayer
    # version if you would prefer to use mplayer instead of VLC
    cmdline = ['vlc', '--demux', 'h264', '-']
    #cmdline = ['mplayer', '-fps', '25', '-cache', '1024', '-']
    player = subprocess.Popen(cmdline, stdin=subprocess.PIPE)
    while True:
        # Repeatedly read 1k of data from the connection and write it to
        # the media player's stdin
        data = connection.read(1024)
        if not data:
            break
        player.stdin.write(data)
finally:
    connection.close()
    server_socket.close()
    player.terminate()
```

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

You will probably notice several seconds of latency with this setup. This is normal and is because media players buffer several seconds to guard against unreliable network streams. Some media players (notably mplayer in this case) permit the user to skip to the end of the buffer (press the right cursor key in mplayer), reducing the latency by increasing the risk that delayed / dropped network packets will interrupt the playback.

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

```
import socket
import time
import picamerax

# 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 = picamerax.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 picamerax

camera = picamerax.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).

3.14 Overlaying images on the preview

The camera preview system can operate multiple layered renderers simultaneously. While the picamerax 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 21).

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

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

```
import picamerax
from PIL import Image
from time import sleep

camera = picamerax.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 picamerax
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 = picamerax.PiCamera()
camera.resolution = (1280, 720)
camera.framerate = 24
camera.start_preview()
# Add the overlay directly into layer 3 with transparency;
```

(continues on next page)

(continued from previous page)

```
# we can omit the size parameter of add_overlay as the
# size is the same as the camera's resolution
o = camera.add_overlay(memoryview(a), layer=3, alpha=64)
try:
    # Wait indefinitely until the user terminates the script
    while True:
        time.sleep(1)
finally:
    camera.remove_overlay(o)
```

Note: The above example works in Python 3.x only. In Python 2.7, `memoryview()` lacks the necessary interface to work with overlays; use `np.getbuffer(a)` instead of `memoryview(a)`.

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

New in version 1.8.

3.15 Overlaying text on the output

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

```
import picamerax
import time

camera = picamerax.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 picamerax
import time
import itertools

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

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

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

```
import picamerax
import datetime as dt

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

3.16 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[20].

However, provided you have the RPi.GPIO[21] package installed, and provided your Python process is running with sufficient privileges (typically this means running as root with `sudo python`), you can also control the LED via the `led` attribute:

```
import picamerax

camera = picamerax.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.

[20] https://www.raspberrypi.org/documentation/configuration/config-txt.md
[21] 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 `picamerax.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).

4.1 Capturing to a numpy array

Since 1.11, picamerax can capture directly to any object which supports Python's buffer protocol (including numpy's `ndarray`[22]). 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`[23] 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`[24] (Python 3.x only):

```
import time
import picamerax
import numpy as np

with picamerax.PiCamera() as camera:
    camera.resolution = (320, 240)
    camera.framerate = 24
    time.sleep(2)
```

(continues on next page)

[22] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[23] https://docs.python.org/3.5/library/functions.html#bytes
[24] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

(continued from previous page)

```
    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 picamerax
import numpy as np

with picamerax.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.

4.2 Capturing to an OpenCV object

This is a variation on *Capturing to a numpy array* (page 23). OpenCV[25] 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 picamerax
import numpy as np
import cv2

with picamerax.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.

4.3 Unencoded image capture (YUV format)

If you want images captured without loss of detail (due to JPEG's lossy compression), you are probably better off exploring PNG as an alternate image format (PNG uses lossless compression). However, some applications (par-

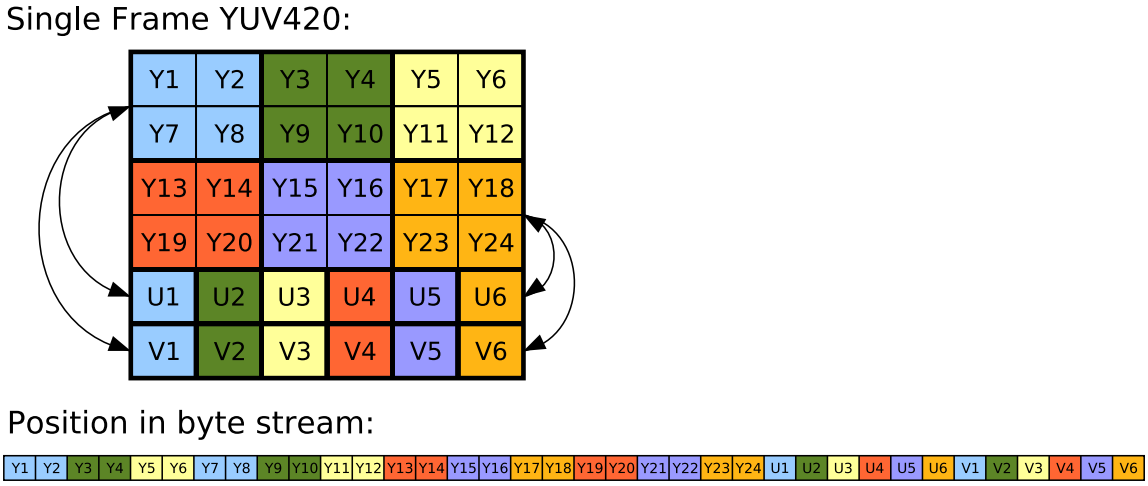
[25] http://opencv.org/

ticularly scientific ones) simply require the image data in numeric form. For this, the `'yuv'` format is provided:

```
import time
import picamerax

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

The specific YUV[26] format used is YUV420[27] (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[28] 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[29] arrays, and converting the data to RGB format in an efficient manner:

```
from __future__ import division
```

(continues on next page)

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

(continued from previous page)

```
import time
import picamerax
import numpy as np

width = 100
height = 100
stream = open('image.data', 'w+b')
# Capture the image in YUV format
with picamerax.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()`[30] in the code above instead of `io.open()`[31] as in the other examples. This is because numpy's `numpy.fromfile()`[32] method annoyingly only accepts "real" file objects.

This recipe is now encapsulated in the `PiYUVArray` class in the *picamerax.array* (page 107) module, which means the same can be achieved as follows:

```
import time
import picamerax
import picamerax.array

with picamerax.PiCamera() as camera:
    with picamerax.array.PiYUVArray(camera) as stream:
        camera.resolution = (100, 100)
```

(continues on next page)

[30] https://docs.python.org/3.5/library/functions.html#open
[31] https://docs.python.org/3.5/library/io.html#io.open
[32] https://numpy.org/doc/stable/reference/generated/numpy.fromfile.html#numpy.fromfile

(continued from previous page)

```
        camera.start_preview()
        time.sleep(2)
        camera.capture(stream, 'yuv')
        # Show size of YUV data
        print(stream.array.shape)
        # Show size of RGB converted data
        print(stream.rgb_array.shape)
```

As of 1.11 you can also capture directly to numpy arrays (see *Capturing to a numpy array* (page 23)). Due to the difference in resolution of the Y and UV components, this isn't directly useful (if you need all three components, you're better off using `PiYUVArray` as this rescales the UV components for convenience). However, if you only require the Y plane you can provide a buffer just large enough for this plane and ignore the error that occurs when writing to the buffer (picamerax will deliberately write as much as it can to the buffer before raising an exception to support this use-case):

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

with picamerax.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 27) for a method of having the camera output RGB data directly.

Note: Capturing so-called "raw" formats (`'yuv'`, `'rgb'`, etc.) does not provide the raw bayer data from the camera's sensor. Rather, it provides access to the image data after GPU processing, but before format encoding (JPEG, PNG, etc). Currently, the only method of accessing the raw bayer data is via the *bayer* parameter to the `capture()` method. See *Raw Bayer data captures* (page 46) for more information.

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

Changed in version 1.5: Added note about new *picamerax.array* (page 107) module.

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

4.4 Unencoded image capture (RGB format)

The RGB format is rather larger than the YUV[33] format discussed in the section above, but is more useful for most analyses. To have the camera produce output in RGB[34] format, you simply need to specify `'rgb'` as the format for the `capture()` method instead:

```
import time
import picamerax
```

(continues on next page)

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

(continued from previous page)

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

The size of RGB[35] data can be calculated similarly to YUV[36] captures. Firstly round the resolution appropriately (see *Unencoded image capture (YUV format)* (page 24) 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[37] 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[38] data are all equally sized (in contrast to YUV420[39]) it is trivial to capture directly into a numpy array (Python 3.x only; see *Capturing to a numpy array* (page 23) for Python 2.x instructions):

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

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

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

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

Changed in version 1.5: Added note about new *`picamerax.array`* (page 107) module.

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

[35] https://en.wikipedia.org/wiki/RGB
[36] https://en.wikipedia.org/wiki/YUV
[37] https://en.wikipedia.org/wiki/RGB
[38] https://en.wikipedia.org/wiki/RGB
[39] https://en.wikipedia.org/wiki/YUV#Y.E2.80.B2UV420p_.28and_Y.E2.80.B2V12_or_YV12.29_to_RGB888_conversion

4.5 Custom outputs

All methods in the picamerax 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`[40]). However, building a custom output object is extremely easy and in certain cases very useful. A file-like object (as far as picamerax 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 45). 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 picamerax

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 picamerax.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 picamerax
import numpy as np

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

class MyMotionDetector(object):
    def __init__(self, camera):
```

(continues on next page)

[40] https://docs.python.org/3.5/library/io.html#io.BytesIO

(continued from previous page)

```
        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 picamerax.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 *picamerax.array* (page 107) module which implement several custom outputs for analysis of data with numpy. In particular, the `PiMotionAnalysis` class can be used to remove much of the boiler plate code from the recipe above:

```
import picamerax
import picamerax.array
import numpy as np

class MyMotionDetector(picamerax.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 picamerax.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.

4.6 Unconventional file outputs

As noted in prior sections, picamerax 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()`[41].
- A *custom output* (page 29).
- 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 picamerax treats files, especially when dealing with "unconventional" files (e.g. pipes, FIFOs, etc.)

When given a filename, picamerax 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 picamerax will presume to close the output for you, because picamerax 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 picamerax 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 picamerax

with picamerax.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')
```

4.7 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 73)).

[41] https://docs.python.org/3.5/library/functions.html#open

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

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

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

```
import time
import picamerax

with picamerax.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 picamerax

frames = 60

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

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

```
import time
import picamerax

frames = 60

def filenames():
```

(continues on next page)

(continued from previous page)

```
    frame = 0
    while frame < frames:
        yield 'image%02d.jpg' % frame
        frame += 1

with picamerax.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[42] (`FF D8`). This means we can use a *custom output* (page 29) to separate the frames out of an MJPEG video recording by inspecting the first two bytes of each buffer:

```
import io
import time
import picamerax

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 picamerax.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)))
```

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

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:

```
import io
import time
import threading
import queue
import picamerax

class ImageProcessor(threading.Thread):
    def __init__(self, owner):
        super(ImageProcessor, self).__init__()
        self.terminated = False
        self.owner = owner
        self.start()

    def run(self):
        # This method runs in a separate thread
        while not self.terminated:
            # Get a buffer from the owner's outgoing queue
            try:
                stream = self.owner.outgoing.get(timeout=1)
            except queue.Empty:
                pass
            else:
                stream.seek(0)
                # Read the image and do some processing on it
                #Image.open(stream)
                #...
                #...
                # Set done to True if you want the script to terminate
                # at some point
                #self.owner.done=True
                stream.seek(0)
                stream.truncate()
                self.owner.incoming.put(stream)

class ProcessOutput(object):
    def __init__(self, threads):
        self.done = False
        # Construct a pool of image processors, a queue of incoming buffers,
        # and a (currently empty) queue of outgoing buffers. Prime the incoming
        # queue with proc+1 buffers (+1 to permit output to be written while
        # all procs are busy with existing buffers)
        self.incoming = queue.Queue(threads)
        self.outgoing = queue.Queue(threads)
        self.pool = [ImageProcessor(self) for i in range(threads)]
        buffers = (io.BytesIO() for i in range(threads + 1))
        for buf in buffers:
            self.incoming.put(buf)
        self.buffer = None

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            # New frame; push current buffer to the outgoing queue and attempt
            # to get a buffer from the incoming queue
            if self.buffer is not None:
                self.outgoing.put(self.buffer)
                try:
                    self.buffer = self.incoming.get_nowait()
```

(continues on next page)

(continued from previous page)

```
                except queue.Empty:
                    # No buffers available (means all threads are busy); skip
                    # this frame
                    self.buffer = None
            if self.buffer is not None:
                self.buffer.write(buf)

    def flush(self):
        # When told to flush (this indicates end of recording), shut
        # down in an orderly fashion. Tell all the processor's they're
        # terminated and wait for them to quit
        for proc in self.pool:
            proc.terminated = True
        for proc in self.pool:
            proc.join()

with picamerax.PiCamera(resolution='VGA') as camera:
    camera.start_preview()
    time.sleep(2)
    output = ProcessOutput(4)
    camera.start_recording(output, format='mjpeg')
    while not output.done:
        camera.wait_recording(1)
    camera.stop_recording()
```

4.8 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 29) (via the classes from *picamerax.array* (page 107)), we can produce a fairly rapid color detection script:

```
import picamerax
import numpy as np
from picamerax.array import PiRGBAnalysis
from picamerax.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'
```

(continues on next page)

(continued from previous page)

```
        # If the color has changed, update the display
        if c != self.last_color:
            self.camera.annotate_text = c
            self.last_color = c

with picamerax.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()
```

4.9 Rapid capture and streaming

Following on from *Rapid capture and processing* (page 31), we can combine the video capture technique with *Capturing to a network stream* (page 13). The server side script doesn't change (it doesn't really care what capture technique is being used - it just reads JPEGs off the wire). The changes to the client side script can be minimal at first - just set *use_video_port* to `True` in the `capture_continuous()` call:

```
import io
import socket
import struct
import time
import picamerax

client_socket = socket.socket()
client_socket.connect(('my_server', 8000))
connection = client_socket.makefile('wb')
try:
    with picamerax.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)
```

(continues on next page)

(continued from previous page)

```
            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 31) we can manage much faster:

```
import io
import socket
import struct
import time
import picamerax

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 picamerax.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.

4.10 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[43] 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`[44] module to make a simple video streaming server:

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

PAGE="""\
<html>
<head>
<title>picamerax 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')
```

(continues on next page)

[43] https://github.com/waveform80/pistreaming/
[44] https://docs.python.org/3.5/library/http.server.html#module-http.server

(continued from previous page)

```
            self.send_header('Content-Length', len(content))
            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 picamerax.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)

4.11 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 31)) 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 picamerax

with picamerax.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).

4.12 Recording at multiple resolutions

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

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

```
import picamerax

with picamerax.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.

4.13 Recording motion vector data

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

```
import picamerax

with picamerax.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.framerate = 30
```

(continues on next page)

(continued from previous page)

```
    camera.start_recording('motion.h264', motion_output='motion.data')
    camera.wait_recording(10)
    camera.stop_recording()
```

Motion data is calculated at the macro-block[45] 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[46]) 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[47] with Pythagoras' theorem. The SAD (Sum of Absolute Differences[48]) 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

```

(continues on next page)

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

(continued from previous page)

```
width = 640
height = 480
cols = (width + 15) // 16
cols += 1
rows = (height + 15) // 16

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

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

You may wish to investigate the `PiMotionArray` and `PiMotionAnalysis` classes in the *picamerax.array* (page 107) module which simplifies the above recipes to the following:

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

with picamerax.PiCamera() as camera:
    with picamerax.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 picamerax
from picamerax.array import PiMotionAnalysis
```

(continues on next page)

(continued from previous page)

```
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 picamerax.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.

4.14 Splitting to/from a circular stream

This example builds on the one in *Recording to a circular stream* (page 16) and the one in *Capturing images whilst recording* (page 39) to demonstrate the beginnings of a security application. As before, a `PiCameraCircularIO` instance is used to keep the last few seconds of video recorded in memory. While the video is being recorded, video-port-based still captures are taken to provide a motion detection routine with some input (the actual motion detection algorithm is left as an exercise for the reader).

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

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

This example also demonstrates using the *seconds* parameter of the `copy_to()` method to limit the before file to 10 seconds of data (given that the circular buffer may contain considerably more than this).

New in version 1.0.

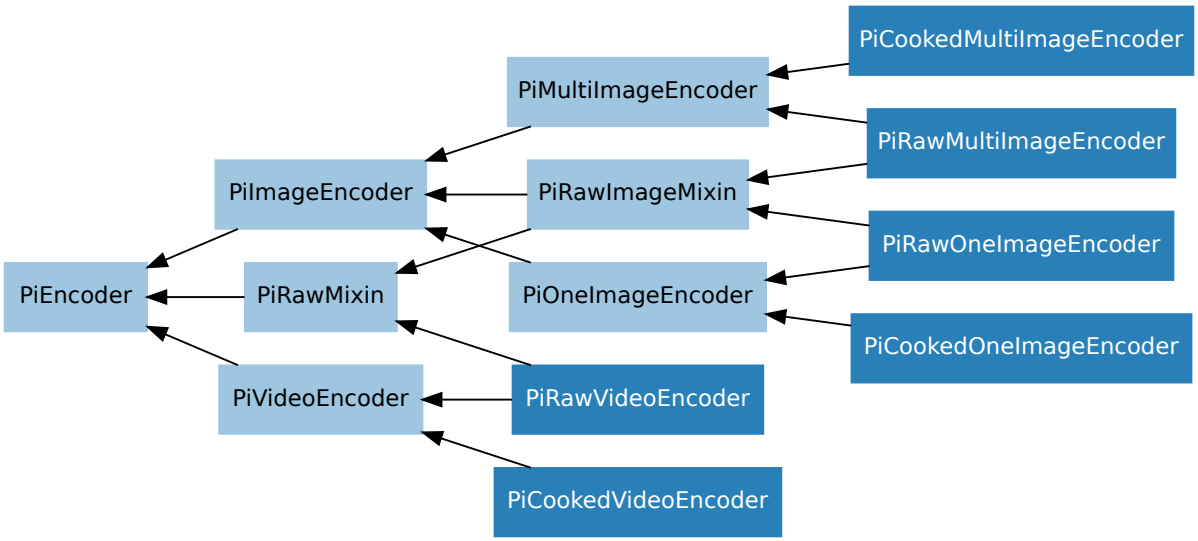
Changed in version 1.11: Added use of `copy_to()`

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

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



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

Method(s)	Calls	Returns

`capture()`
`capture_continuous()`
`capture_sequence()`
	`_get_image_encoder()`	`PiCookedOneImageEncoder`
		`PiRawOneImageEncoder`

`capture_sequence()`	`_get_images_encoder()`	`PiCookedMultiImageEncoder` `PiRawMultiImageEncoder`
`start_recording()` `record_sequence()`	`_get_video_encoder()`	`PiCookedVideoEncoder` `PiRawVideoEncoder`

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

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

```
import picamerax
import picamerax.mmal as mmal

# Override PiVideoEncoder to keep track of the number of each type of frame
class MyEncoder(picamerax.PiCookedVideoEncoder):
```

(continues on next page)

(continued from previous page)

```
    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(picamerax.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 40). 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.

4.16 Raw Bayer data captures

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

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

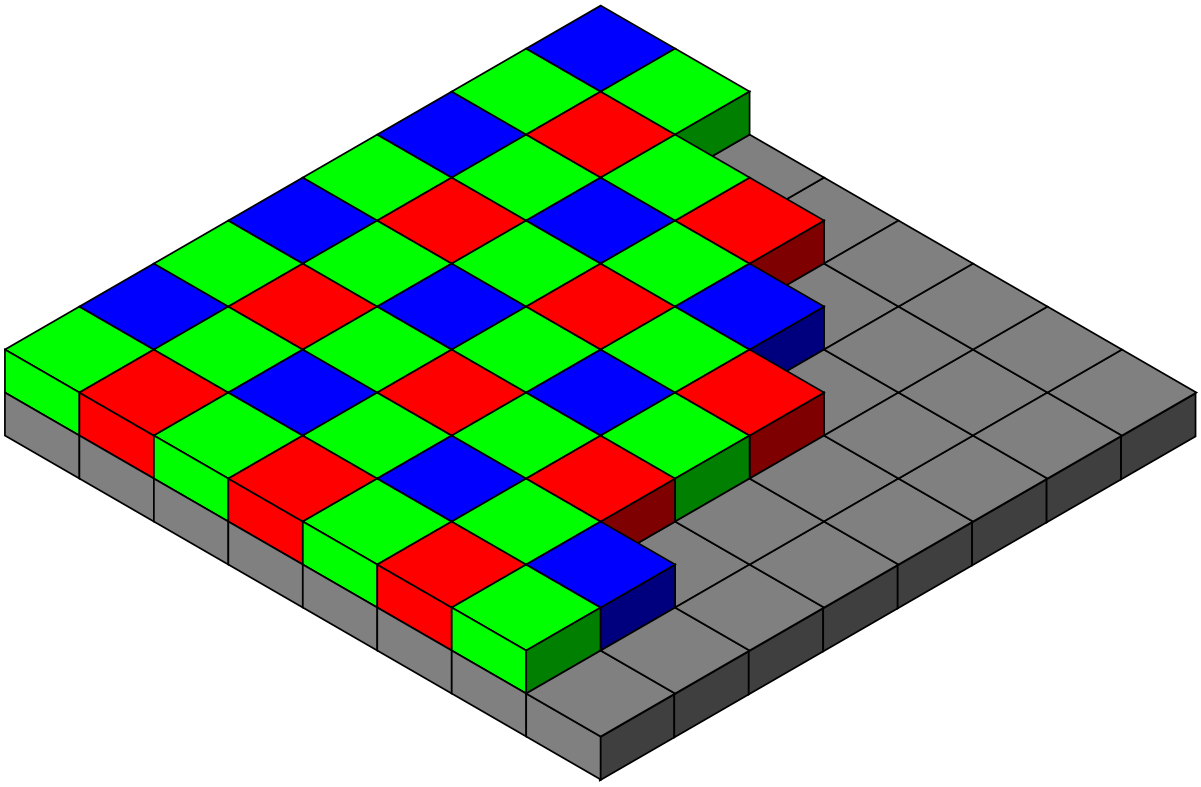
Raw Bayer data differs considerably from simple unencoded captures; it is the data recorded by the camera's sensor prior to *any* GPU processing including auto white balance, vignette compensation, smoothing, down-scaling, etc. This also means:

- Bayer data is *always* full resolution, regardless of the camera's output `resolution` and any `resize` parameter.

- Bayer data occupies the last 6,404,096 bytes of the output file for the V1 module, or the last 10,270,208 bytes for the V2 module. The first 32,768 bytes of this is header data which starts with the string `'BRCM'`.
- Bayer data consists of 10-bit values, because this is the sensitivity of the OV5647[49] and IMX219[50] sensors used in the Pi's camera modules. The 10-bit values are organized as 4 8-bit values, followed by the low-order 2-bits of the 4 values packed into a fifth byte.

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

- Bayer data is organized in a BGGR pattern (a minor variation of the common Bayer CFA[51]). 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[52] at the very least, and probably some form of color balance[53].

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[54] 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

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

slower than normal image captures:

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


import io
import time
import picamerax
import numpy as np
from numpy.lib.stride_tricks import as_strided

stream = io.BytesIO()
with picamerax.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 DDCCBBAA
```

(continues on next page)

(continued from previous page)

```
#
# 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] >> (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, [
```

(continues on next page)

(continued from previous page)

```
    (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` class in the *picamerax.array* (page 107) module, which means the same can be achieved as follows:

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

with picamerax.PiCamera() as camera:
    with picamerax.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 *picamerax.array* (page 107) module.

4.17 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[55]). This will fire before (flash metering[56]) 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[57]. 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`
Raspberry Pi Zero Wireless rev 1.1	`/videocore/pins_pi0w`

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[58] (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:

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

```
$ 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
- `-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 picamerax

with picamerax.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";
```

(continues on next page)

(continued from previous page)

```
    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)

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

You've named your script `picamerax.py` (or you've named some other script `picamerax.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.

5.2 Can I put the preview in a window?

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

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

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

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

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

5.4 The preview doesn't work on my PiTFT screen

The camera's preview system directly overlays the Pi's output on the HDMI or composite video ports. At this time, it will not operate with GPIO-driven displays like the PiTFT. Some projects, like the Adafruit Touchscreen Camera project[59], have approximated a preview by rapidly capturing unencoded images and displaying them on the PiTFT instead.

5.5 How much power does the camera require?

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

```
import picamerax
import time

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

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

```
import picamerax
import time

for i in range(60):
    with picamerax.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 12).

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.

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

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

See the *Capturing consistent images* (page 11) recipe.

5.7 Can I use picamerax with a USB webcam?

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

5.8 How can I tell what version of picamerax I have installed?

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

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

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

If you are using Raspbian, firstly check that you haven't got both a PyPI (`pip`) and an apt (`apt-get`) installation of picamerax 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 picamerax 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 picamerax packages become accessible in the Raspbian repository.

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

5.10 Why is there so much latency when streaming video?

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

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

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

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

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

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

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

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

Read the note at the bottom of the *Recording to a circular stream* (page 16) recipe. When you set the number of seconds for the circular stream you are setting a *lower bound* for a given bitrate (which defaults to 17Mbps - the same as the video recording default). If the recorded scene has low motion or complexity the stream can store considerably more than the number of seconds specified.

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

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

5.12 Can I move the annotation text?

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

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

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

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

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

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 \
```

5.14 Out of resources at full resolution on a V2 module

See *Hardware Limits* (page 77).

5.15 Preview flickers at full resolution on a V2 module

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

```
from picamerax import PiCamera

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

5.16 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`[61] module creates multiple copies of a Python process (usually via `os.fork()`[62]) for the purpose of parallel processing. Whilst you can use `multiprocessing`[63] with picamerax, you must ensure that only a *single* process creates a `PiCamera` instance at any given time.

[61] https://docs.python.org/3.5/library/multiprocessing.html#module-multiprocessing
[62] https://docs.python.org/3.5/library/os.html#os.fork
[63] https://docs.python.org/3.5/library/multiprocessing.html#module-multiprocessing

The following script demonstrates an approach with one process that owns the camera, which handles disseminating captured frames to other processes via a Queue[64]:

```
import os
import io
import time
import multiprocessing as mp
from queue import Empty
import picamerax
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 picamerax.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)
```

(continues on next page)

[64] https://docs.python.org/3.5/library/multiprocessing.html#multiprocessing.Queue

(continued from previous page)

```
        ]
    for proc in processing_procs:
        proc.start()
    capture_proc.start()
    for proc in processing_procs:
        proc.join()
    capture_proc.join()
```

5.17 VLC won't play back MJPEG recordings

MJPEG[65] is a particularly ill-defined format (see "Disadvantages[66]") 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 38) 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
```

[65] https://en.wikipedia.org/wiki/Motion_JPEG
[66] 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 picamerax uses.

6.1 Theory of Operation

Many questions I receive regarding picamerax 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[67] model, presenting first a technically inaccurate but useful model of the camera's operation, then refining it closer to the truth later on.

6.1.1 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[68]) 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[69] 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".

6.1.2 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.

[67] https://en.wikipedia.org/wiki/Lie-to-children
[68] https://en.wikipedia.org/wiki/Digital_single-lens_reflex_camera
[69] 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 picamerax 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).

6.1.3 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[70]. 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:

[70] 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` 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` 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`.

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

6.1.4 Sensor gain

The other important factor influencing sensor element counts, aside from line read-out time, is the sensor's gain[71]. Specifically, the gain given by the `analog_gain` attribute (the corresponding `digital_gain` 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[72] (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).

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

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[73] 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 picamerax, but various attributes can be used to "influence" it.

- Setting `exposure_mode` 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` 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).
  - `'night'` is intended as a stills mode, so it permits very long exposure times while attempting to keep gains low.
- The `iso` 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[74] standard).

  Hence, one might be tempted to think that `iso` provides a means of fixing the gains, but this isn't entirely true: the `exposure_mode` 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).

6.1.5 Division of labor

At this point, a reader familiar with operating system theory may be questioning how a non real-time operating system[75] (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

[73] https://en.wikipedia.org/wiki/Electromagnetic_spectrum
[74] https://en.wikipedia.org/wiki/Film_speed#Current_system:_ISO
[75] https://en.wikipedia.org/wiki/Real-time_operating_system

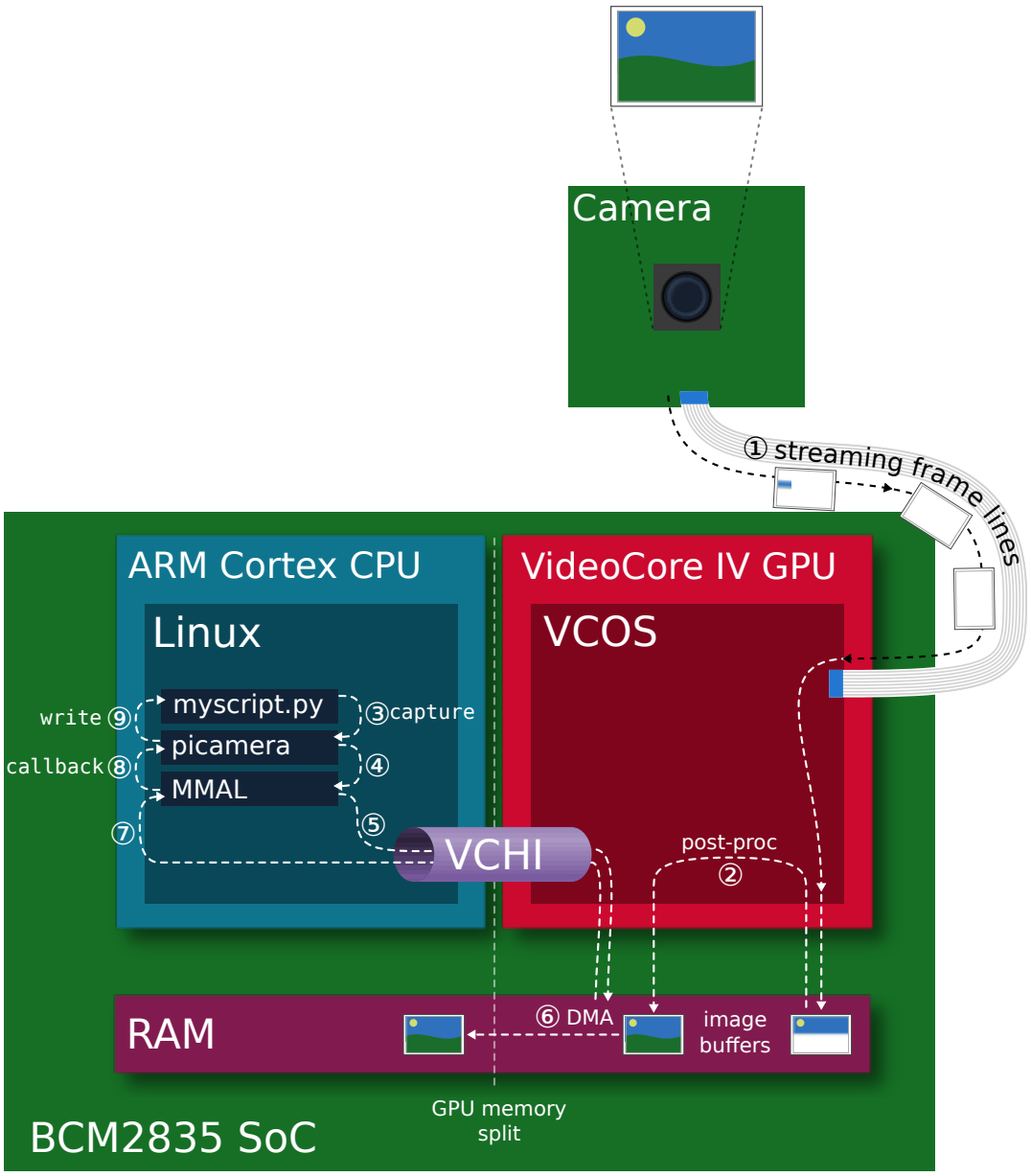
the user doesn't directly interact with it anyway, it is perhaps simpler to think of the GPU as running something called VCOS (without thinking too much about what that actually is).

The following diagram illustrates that the BCM2835 system on a chip[76] (SoC) is comprised of an ARM Cortex CPU running Linux (under which is running `myscript.py` which is using picamerax), 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 picamerax.
4. The picamerax 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).
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[77] 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 picamerax library, which in turn retrieves the frame (in reality, this requires more MMAL and VCHI activity).
9. Finally, picamerax calls `write` on the output object provided by `myscript.py`.

[76] https://en.wikipedia.org/wiki/System_on_a_chip
[77] https://en.wikipedia.org/wiki/Direct_memory_access



6.1.6 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.

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[78] 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`, `vflip`, and `sensor_mode`.
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.

[78] https://andor.oxinst.com/learning/view/article/ccd-binning

3. Next the GPU's image signal processor[79] (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[80]. See `awb_gains` and `awb_mode`.
   - **Digital gain**: As mentioned above, this is a straight-forward post-processing step that applies a gain to the Bayer values[81]. See `digital_gain`.
   - **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[82] 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` and `video_denoise`.
   - **Sharpening**: An algorithm to enhance edges in the image. See `sharpness`.
   - **Color processing**: The `brightness`, `contrast`, and `saturation` 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[83]; 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). Firstly, the zoom is applied (see `zoom`) and then the image is resized to the requested resolution (see `resolution`).

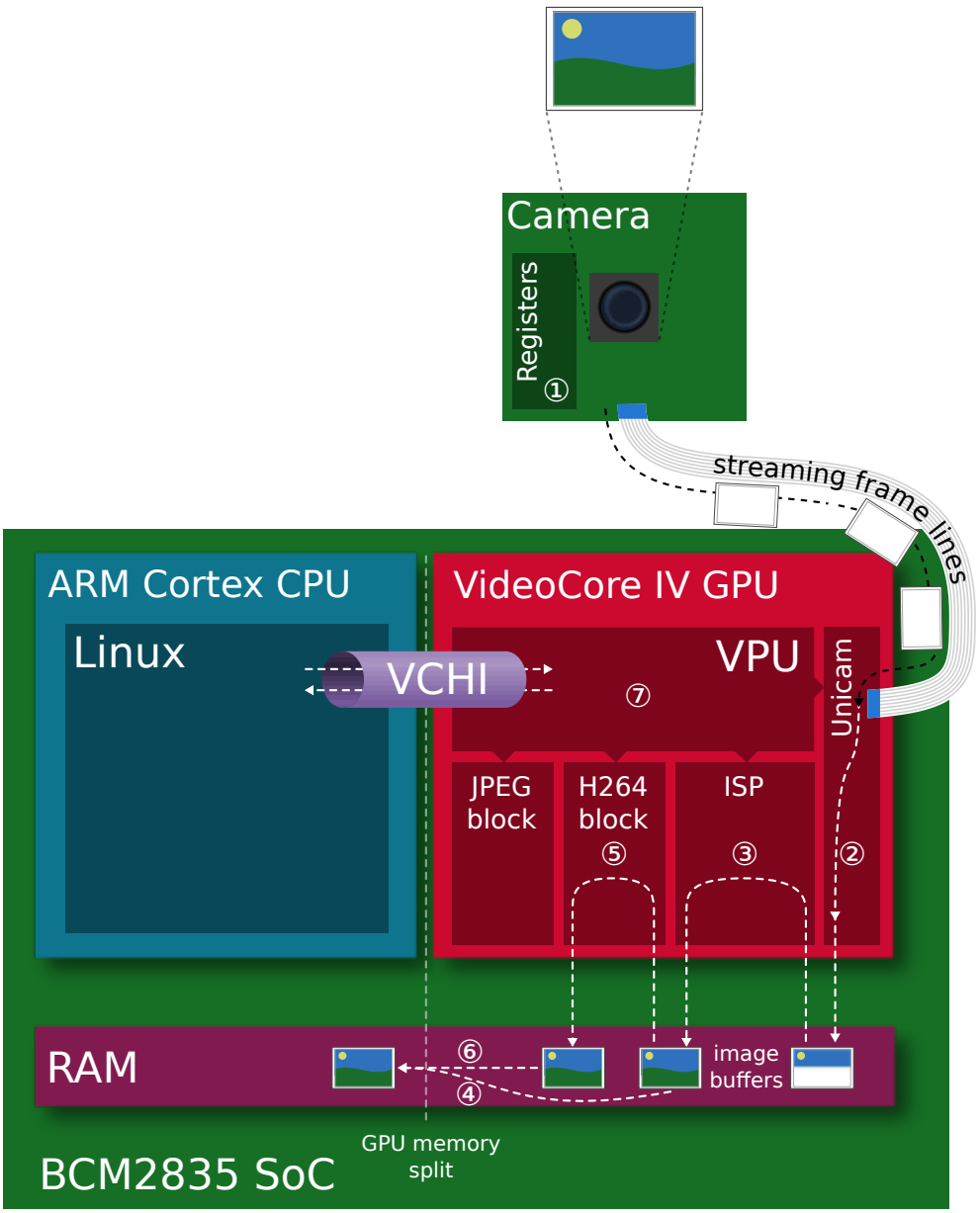
   Some of these steps can be controlled directly (e.g. brightness, noise reduction), others can only be influenced (e.g. analog and digital gain), and the remainder are not user-configurable at all (e.g. demosaic and lens shading).

   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[84]. The ISP might be used to convert to RGB, but that's all.
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[85] (DCTs), while the H264 block will include hardware for performing motion estimation[86].
6. Once encoded, the output is copied to the CPU via DMA[87].

[79] https://en.wikipedia.org/wiki/Image_processor
[80] https://en.wikipedia.org/wiki/Color_balance
[81] https://en.wikipedia.org/wiki/Bayer_filter
[82] https://en.wikipedia.org/wiki/YUV#Y.E2.80.B2UV420p_.28and_Y.E2.80.B2V12_or_YV12.29_to_RGB888_conversion
[83] https://en.wikipedia.org/wiki/Fisheye_lens
[84] https://en.wikipedia.org/wiki/Direct_memory_access
[85] https://en.wikipedia.org/wiki/Discrete_cosine_transform
[86] https://en.wikipedia.org/wiki/Motion_estimation
[87] https://en.wikipedia.org/wiki/Direct_memory_access

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[88].



6.1.7 Feedback loops

There are a couple of feedback loops running within the process described above:

1. When `exposure_mode` is not `'off'`, automatic gain control (AGC) gathers statistics from each frame (prior to the de-mosaic phase in the ISP, step 3 in the previous diagram). The AGC tweaks the analog and digital gains, and the exposure time (line read-out time), attempting to nudge subsequent frames towards a target Y (luminance[89]) value.

2. When `awb_mode` is not `'off'`, automatic white balance (AWB) gathers statistics from frames (again, prior to the de-mosaic phase). Typically AWB analysis only occurs on 1 out of every 3 streamed frames because it is computationally expensive. It adjusts the red and blue gains (`awb_gains`), attempting to

[88] https://github.com/hermanhermitage/videocoreiv
[89] https://en.wikipedia.org/wiki/Relative_luminance

nudge subsequent frames towards the expected color balance[90].

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` 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` 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` 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[91]) 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` attribute, which is measured in increments of 1/6th of an f-stop[92]. So if, whilst the exposure time is fixed, you increase the luminance that the camera is aiming for by a couple of stops and 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` 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
```

6.2 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:

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

#	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[93]
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[94]	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.

Modes with full field of view[95] (FoV) capture images 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 images just from the center of the sensor. The combination of FoV limiting, and binning[96] is used to achieve the requested resolution.

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

[93] In fact, the sensor uses a 2x2 binning in combination with a 2x2 skip to achieve the equivalent of a 4x4 reduction in resolution.

[94] Sensor mode 3 on the V2 module appears to be a duplicate of sensor mode 2, 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.

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

[96] https://andor.oxinst.com/learning/view/article/ccd-binning

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

You can manually select the sensor's mode with the *sensor_mode* parameter in the `PiCamera` 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` and `framerate`. 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` is to the mode's resolution, the better. Downscaling from a higher sensor resolution to a lower output resolution is preferable to upscaling from a lower sensor resolution to a higher output resolution.
- The requested `framerate` should be within the range of the sensor mode.
- The closer the aspect ratio of the requested `resolution` to the mode's resolution, the better. Attempts to set resolutions with aspect ratios other than 4:3 or 16:9 (which are the only ratios directly supported by the modes in the tables above), result in the selection of the mode which maximizes the resulting field of view[97] (FoV).

Here are a few examples for the V1 camera module to clarify the operation of this process:

- If you set the `resolution` to 1024x768 (a 4:3 aspect ratio), and the `framerate` 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` to 1280x720 (a 16:9 wide-screen aspect ratio), and the `framerate` to anything less than 49fps, the 1296x730 mode (5) will be selected and downscaled appropriately.

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

- Setting the `resolution` to 1920x1080 and the `framerate` 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` of 800x600 and a `framerate` of 60fps will select the 640x480 60fps mode, even though it requires upscaling because the algorithm considers the framerate to take precedence in this case.
- Any attempt to capture an image without using the video port will (temporarily) select the 2592x1944 mode while the capture is performed (this is what causes the flicker you sometimes see when a preview is running while a still image is captured).

Resolution	Framerate (fps)	Result
1024x768 (a 4:3 aspect ratio)	< 42	The 1296x972 mode (4) will be selected, and the GPU will downscale the result to 1024x768.
1280x720 (a 16:9 wide-screen aspect ratio)	< 49	The 1296x730 mode (5) will be selected and downscaled appropriately.
1920x1080	30	This 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.
800x600	60	This selects 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).

6.3 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). If you encounter "out of resources" errors when attempting long-exposure captures with a V2 module, increase `gpu_mem` in `/boot/config.txt`.
- The maximum resolution of the V2 camera can also cause issues with previews. Currently, picamerax runs previews at the same resolution as captures (equivalent to `-fp` in `raspistill`). To achieve full resolution operation with the V2 camera module, you may need to increase `gpu_mem` in `/boot/config.txt`, or configure the preview to use a lower `resolution` 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` before setting `shutter_speed`.

6.4 MMAL

The MMAL layer below picamerax 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 picamerax and how they influence the camera's behaviour.

6.4.1 The Still Port

Whenever the still port is used to capture images, it (briefly) forces the camera's mode to one of the two supported still modes (see *Sensor Modes* (page 73)), meaning 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()` methods when their *use_video_port* parameter is `False` (which it is by default).

6.4.2 The Video Port

The video port is somewhat simpler in that it never changes the camera's mode. The video port is used by the `start_recording()` method (for recording video), and also by the various `capture()` methods when their *use_video_port* parameter is `True`. Images captured from the video port tend to have a "grainy" appearance, much more akin to a video frame than the images captured by the still port. This is because the still port uses a stronger noise reduction algorithm than the video port.

6.4.3 The Preview Port

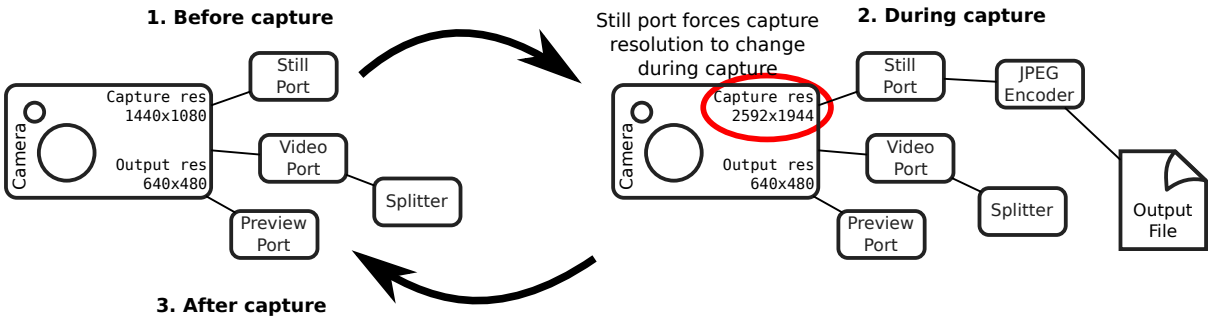
The preview port operates more or less identically to the video port. The preview port is always connected to some form of output to ensure that the auto-gain algorithm can run. When an instance of `PiCamera` is constructed, the preview port is initially connected to an instance of `PiNullSink`. When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

6.4.4 Pipelines

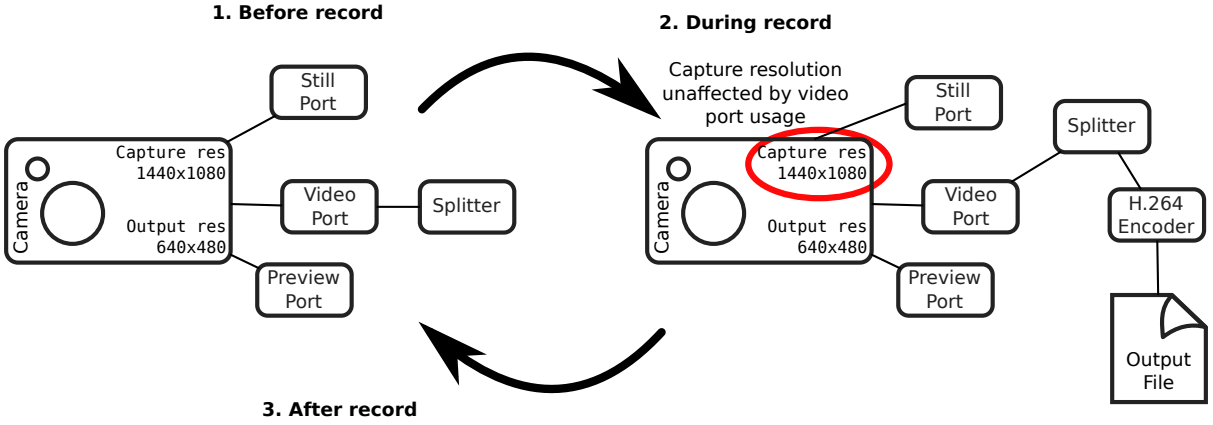
This section provides some detail about the MMAL pipelines picamerax constructs in response to various method calls.

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).

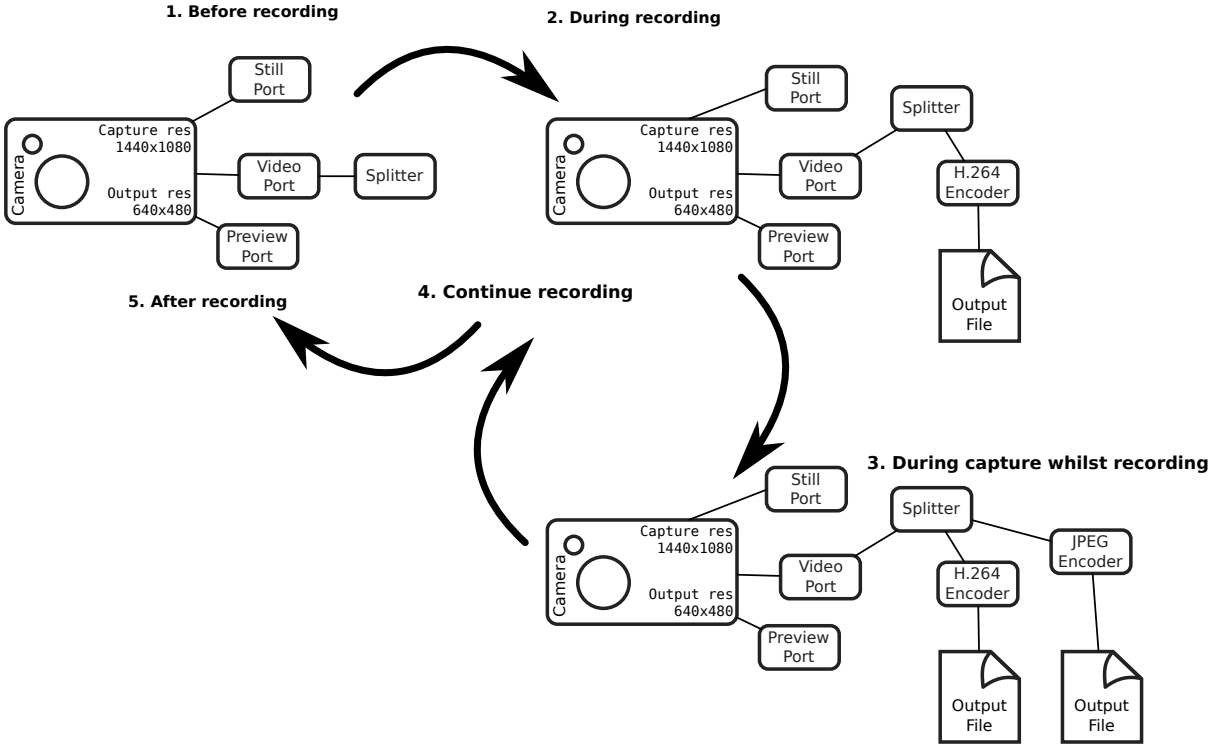
Encoders are connected directly to the still port. For example, when capturing a picture using the still port, the camera's state conceptually moves through these states:



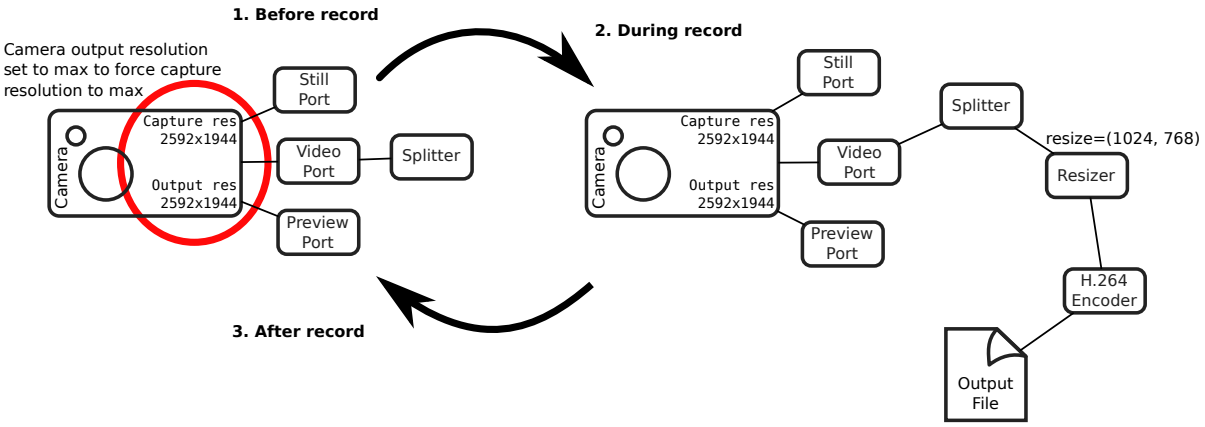
As you have probably noticed in the diagram above, the video port is a little more complex. In order to permit simultaneous video recording and image capture via the video port, a "splitter" component is permanently connected to the video port by picamerax, and encoders are in turn attached to one of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



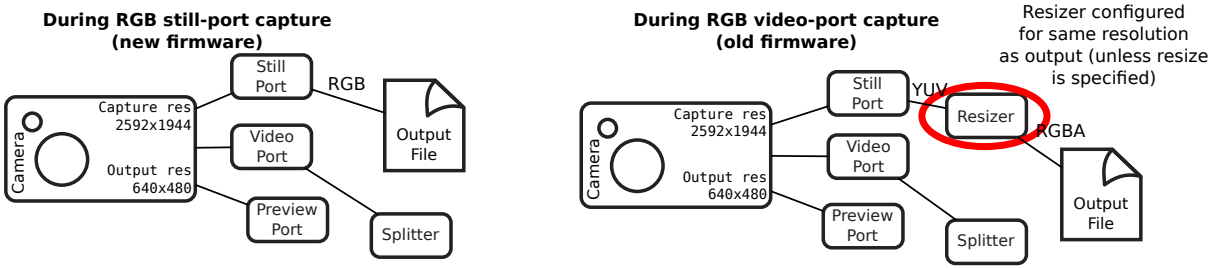
When the `resize` parameter is passed to one of the methods above, a resizer component is placed between the camera's ports and the encoder, causing the output to be resized before it reaches the encoder. This is particularly useful for video recording, as the H.264 encoder cannot cope with full resolution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



Finally, when performing unencoded captures an encoder is obviously not required. Instead, data is taken directly from the camera's ports. However, various firmware limitations require adjustments within the pipeline in order to achieve the 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), but they can be configured for YUV output. So in this case, picamerax configures the still port for YUV output, attaches a 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 picamerax will simply configure the still port for RGB output (since 1.11):



6.4.5 Encodings

The ports used to connect MMAL components together pass image data around in particular encodings. Often, this is the YUV420[98] encoding, this is the "preferred" internal format for the pipeline, and on rare occasions it's RGB[99] (RGB is a large and rather inefficient format). However, there is another format available, called 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. These are 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.

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

- 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`) 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`, not currently used by picamerax's high level API, but available from the *mmalobj* (page 109) layer) accepts OPAQUE input, and will produce almost any unencoded output, including YUV, RGB, BGR, RGBA, and BGRA, but not OPAQUE.

The *mmalobj* (page 109) layer, introduced in picamerax 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 - link). However, as MMAL is the API under-pinning the picamerax library (along with the official `raspistill` and `raspivid` applications) it is worth understanding.

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

CHAPTER 7

Development

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

https://github.com/waveform80/picamerax

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` class' source, to get a handle on using the *mmalobj* (page 109) layer. This is a layer introduced in picamerax 1.11 to ease the usage of `libmmal` (the underlying library that picamerax, `raspistill`, and `raspivid` all rely upon).

Beneath *mmalobj* (page 109) is a `ctypes`[100] 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 picamerax).

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

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

7.1 Development installation

If you wish to develop picamerax 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 picamerax
$ workon picamerax
(picamerax) $ git clone https://github.com/waveform80/picamerax.git
(picamerax) $ cd picamerax
(picamerax) $ make develop
```

[100] https://docs.python.org/3.5/library/ctypes.html#module-ctypes

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

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

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

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

7.2 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 python-sphinx
```

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

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

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

7.3 Test suite

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

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

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

Furthermore, to distinguish any release which is backwards incompatible, I would increment the major version number in accordance with semantic versioning[101]. 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 picamerax library. All deprecated functionality is documented, and the documentation includes pointers to the intended replacement functionality (see `raw_format` for example). However, Python also provides excellent methods for determining automatically whether any deprecated functionality is being used via the `warnings`[102] module.

8.1 Finding and fixing deprecated usage

As of release 1.8, all deprecated functionality will raise `DeprecationWarning`[103] 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:

[101] http://semver.org/
[102] https://docs.python.org/3.5/library/warnings.html#module-warnings
[103] https://docs.python.org/3.5/library/exceptions.html#DeprecationWarning

```
import io
import time
import picamerax

with picamerax.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 picamerax 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`[104]; "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 picamerax

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

with picamerax.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/picamerax/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/picamerax/camera.py:3125: DeprecationWarning: PiCamera.preview_
↪fullscreen is deprecated; use PiCamera.preview.fullscreen instead
  'PiCamera.preview_fullscreen is deprecated; '
/usr/share/pyshared/picamerax/camera.py:3068: DeprecationWarning: PiCamera.preview_
↪alpha is deprecated; use PiCamera.preview.alpha instead
  'PiCamera.preview_alpha is deprecated; use '
/usr/share/pyshared/picamerax/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/picamerax/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/picamerax/camera.py:1827: DeprecationWarning: PiCamera.raw_
↪format is deprecated; use required format directly with capture methods instead
```

(continues on next page)

[104] https://docs.python.org/3.5/library/exceptions.html#DeprecationWarning

(continued from previous page)

```
  '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[105] 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 picamerax

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

with picamerax.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/picamerax/camera.py", line 1888, in _set_framerate
    n, d = to_rational(value)
  File "/usr/share/pyshared/picamerax/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/picamerax/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 picamerax

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

(continues on next page)

[105] https://docs.python.org/3.5/library/exceptions.html#DeprecationWarning

(continued from previous page)

```
with picamerax.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).

8.2 List of deprecated functionality

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

8.2.1 Unencoded captures

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

The deprecated method of taking unencoded captures looks like this:

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

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

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

8.2.2 Recording quality

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

The deprecated method of setting recording quality looks like this:

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

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

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

8.2.3 Fractions as tuples

Several attributes in picamerax expect rational (fractional) values. In early versions of picamerax, 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[106], float[107], `Decimal`[108], and `Fraction`[109]. 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`[110] instance as well, but one modified to permit access as a tuple in order to maintain backward compatibility. This is also deprecated behaviour. The following example demonstrates accessing the `framerate` attribute as a tuple:

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

In such cases, use the standard `numerator`[111] and `denominator`[112] 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`[113] instance to a float[114] for greater convenience:

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

8.2.4 Preview functions

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

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

[106] https://docs.python.org/3.5/library/stdtypes.html#typesnumeric
[107] https://docs.python.org/3.5/library/stdtypes.html#typesnumeric
[108] https://docs.python.org/3.5/library/decimal.html#decimal.Decimal
[109] https://docs.python.org/3.5/library/fractions.html#fractions.Fraction
[110] https://docs.python.org/3.5/library/fractions.html#fractions.Fraction
[111] https://docs.python.org/3.5/library/fractions.html#fractions.Fraction.numerator
[112] https://docs.python.org/3.5/library/fractions.html#fractions.Fraction.denominator
[113] https://docs.python.org/3.5/library/fractions.html#fractions.Fraction
[114] https://docs.python.org/3.5/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` attribute is `None` when the preview is not active. To accomodate this use-case, optional parameters were added to `start_preview()` to provide initial settings for the preview renderer. The following example illustrates the deprecated method of setting preview related attribtues prior to activating the preview:

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

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

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

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

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

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

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

8.2.5 Array stream truncation

In release 1.8, the base `PiArrayOutput` class was changed to derive from `io.BytesIO`[115] in order to add support for seeking, and to improve performance. The prior implementation had been non-seekable, and therefore to accommodate re-use of the stream between captures the `truncate()` method had an unusual side-effect not seen with regular Python streams: after truncation, the position of the stream was set to the new length of the stream. In all other Python streams, the `truncate` method doesn't affect the stream position. The method is overridden in 1.8 to maintain its unusual behaviour, but this behaviour is nonetheless deprecated.

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

[115] https://docs.python.org/3.5/library/io.html#io.BytesIO

```
with picamerax.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 picamerax 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 picamerax.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 picamerax as well (since such streams were non-seekable in prior versions). In this case, call `seekable()` to determine the correct course of action:

```
with picamerax.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)
```

8.2.6 Confusing crop

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

The following example illustrates the deprecated attribute name:

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

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

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

8.2.7 Incorrect ISO capitalisation

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

The following example illustrates the deprecated attribute case:

```
camera.ISO = 100
```

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

```
camera.iso = 100
```

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

8.2.8 Frame types

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

The following code illustrates usage of the deprecated boolean fields:

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

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

```
if camera.frame.frame_type == picamerax.PiVideoFrameType.key_frame:
    handle_keyframe()
elif camera.frame.frame_type == picamerax.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`[117] in the event of an unrecognized frame type):

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

8.2.9 Annotation background color

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

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

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

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

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

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

[117] https://docs.python.org/3.5/library/exceptions.html#KeyError

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
```

8.2.10 Analysis classes use analyze

The various analysis classes in *picamerax.array* (page 107) were adjusted in 1.11 to use `analyze()` (US English spelling) instead of `analyse` (UK English spelling). The following example illustrates the old usage:

```
import picamerax.array

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

This should simply be re-written as:

```
import picamerax.array

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

8.2.11 Positional args for PiCamera

The `PiCamera` class was adjusted in 1.14 to expect keyword arguments on construction. The following used to be accepted (although it was still rather bad practice):

```
import picamerax

camera = picamerax.PiCamera(0, 'none', False, '720p')
```

This should now be re-written as:

```
import picamerax

camera = picamerax.PiCamera(camera_num=0, stereo_mode='none',
                            stereo_decimate=False, resolution='720p')
```

Although if you only wanted to set `resolution` you could simply write this as:

```
import picamerax

camera = picamerax.PiCamera(resolution='720p')
```

8.2.12 Color module

The `picamerax.color` module has now been split off into the colorzero[118] library and as such is deprecated in its entirety. The colorzero[119] library contains everything that the color module used, along with a few enhancements and several bug fixes and as such the transition is expected to be trivial. Look for any imports of the `Color` class:

```
from picamerax import Color

c = Color('green')
```

Replace these with references to `colorzero.Color`[120] instead:

```
from colorzero import Color

c = Color('green')
```

Alternatively, if the `Color` class is being used directly from picamerax itself:

```
import picamerax

camera = picamerax.PiCamera()
c = picamerax.Color('red')
```

In this case add an import for colorzero, and reference the class from there:

```
import picamerax
import colorzero

camera = picamerax.PiCamera()
c = colorzero.Color('red')
```

[118] https://colorzero.readthedocs.io/
[119] https://colorzero.readthedocs.io/
[120] https://colorzero.readthedocs.io/en/latest/api_color.html#colorzero.Color

CHAPTER 9

API - The PiCamera Class

The picamerax library contains numerous classes, but the primary one that all users are likely to interact with is `PiCamera`, documented below. With the exception of the contents of the *picamerax.array* (page 107) module, all classes in picamerax 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 *picamerax.array* (page 107)):

```
import picamerax
```

9.1 PiCamera

9.2 PiVideoFrameType

9.3 PiVideoFrame

9.4 PiResolution

9.5 PiFramerateRange

9.6 PiSensorMode

CHAPTER 10

API - Streams

The picamerax 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).

10.1 PiCameraCircularIO

10.2 CircularIO

10.3 BufferIO

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

11.1 PiRenderer

11.2 PiOverlayRenderer

11.3 PiPreviewRenderer

11.4 PiNullSink

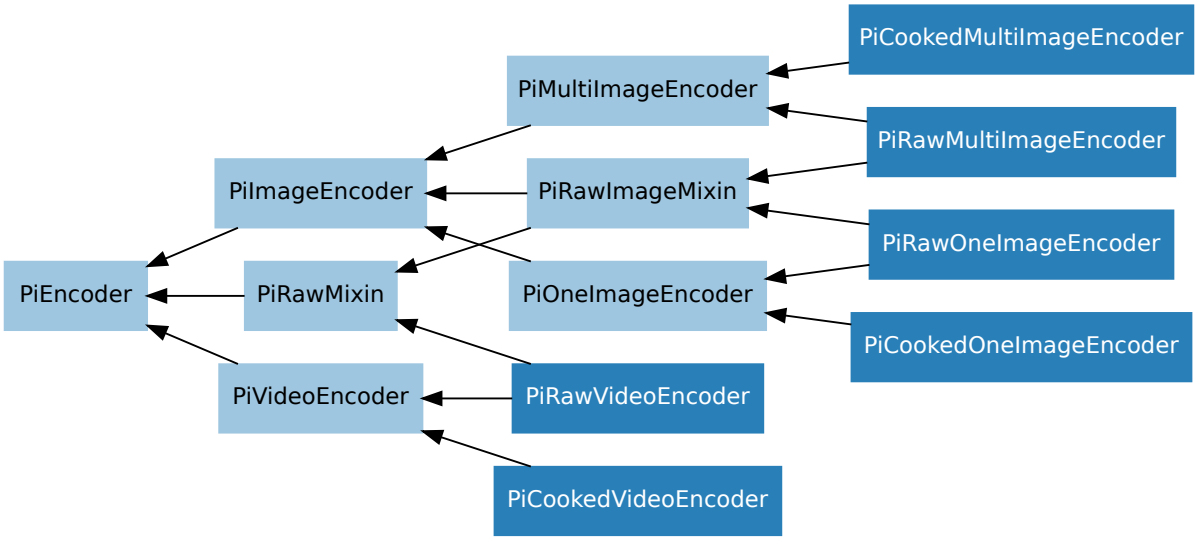
CHAPTER 12

API - Encoders

Encoders are typically used by the camera to compress captured images or video frames for output to disk. However, picamerax 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 45).

Note: It is strongly recommended that you familiarize yourself with the `mmalobj` (page 109) 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:



12.1 PiEncoder

12.2 PiVideoEncoder

12.3 PiImageEncoder

12.4 PiRawMixin

12.5 PiCookedVideoEncoder

12.6 PiRawVideoEncoder

12.7 PiOneImageEncoder

12.8 PiMultiImageEncoder

12.9 PiRawImageMixin

12.10 PiCookedOneImageEncoder

12.11 PiRawOneImageEncoder

12.12 PiCookedMultiImageEncoder

12.13 PiRawMultiImageEncoder

CHAPTER 13

API - Exceptions

All exceptions defined by picamerax are listed in this section. All exception classes utilize multiple inheritance in order to make testing for exception types more intuitive. For example, `PiCameraValueError` derives from both `PiCameraError` and `ValueError`[121]. Hence it will be caught by blocks intended to catch any error specific to the picamerax 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')
```

13.1 Warnings

13.2 Exceptions

13.3 Functions

[121] https://docs.python.org/3.5/library/exceptions.html#ValueError

CHAPTER 14

API - Colors and Color Matching

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

14.1 Color

14.2 Manipulation Classes

CHAPTER 15

API - Arrays

The picamerax library provides a set of classes designed to aid in construction of n-dimensional numpy[122] arrays from camera output. In order to avoid adding a hard dependency on numpy to picamerax, this module (*picamerax.array* (page 107)) is not automatically imported by the main picamerax package and must be explicitly imported, e.g.:

```
import picamerax
import picamerax.array
```

[122] http://www.numpy.org/

15.1 PiArrayOutput

15.2 PiRGBArray

15.3 PiYUVArray

15.4 PiBayerArray

15.5 PiMotionArray

15.6 PiAnalysisOutput

15.7 PiRGBAnalysis

15.8 PiYUVAnalysis

15.9 PiMotionAnalysis

15.10 PiArrayTransform

CHAPTER 16

API - mmalobj

This module provides an object-oriented interface to `libmmal` which is the library underlying picamerax, `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.

16.1 The MMAL Tour

MMAL operates on the principle of pipelines:

- A pipeline consists of one or more MMAL components (`MMALBaseComponent` and derivatives) connected together in series.
- A `MMALBaseComponent` has one or more ports.
- A port (`MMALControlPort` 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 picamerax 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 picamerax only deals with video ports.
  - Ports have a `format` 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` which specifies the number of images expected to be received or sent per second.

  - Video ports also have a `framesize` which specifies the resolution of images/frames accepted or generated by the port.
  - Finally, all ports (control, input and output) have `params` which affect their operation.

- An output port can have a `MMALConnection` 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`.
  - 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`) 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`.
  - 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` 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).

16.1.1 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 109) module with a convenient alias, then construct a `MMALCamera` component, and a `MMALRenderer` component.

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

16.1.2 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 63) 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[123] (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()` call. After the port is committed, note that the buffer size now looks reasonable: 640 * 480 * 1.5 = 460800.

16.1.3 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()` 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()` 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()
```

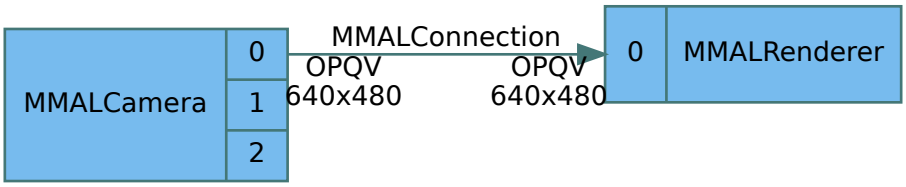
[123] https://en.wikipedia.org/wiki/YUV#Y.E2.80.B2UV420p_.28and_Y.E2.80.B2V12_or_YV12.29_to_RGB888_conversion

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



16.1.4 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.

16.1.5 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)
```

(continues on next page)

(continued from previous page)

```
>>> fx = camera.control.params[mmal.MMAL_PARAMETER_IMAGE_EFFECT]
>>> fx
<picamerax.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/picamerax/picamerax/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`[124], `float`[125], `Fraction`[126], `str`[127], etc. However, some parameters use `ctypes`[128] 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` class, especially property getters and setters.

16.1.6 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()
```

[124] https://docs.python.org/3.5/library/functions.html#int
[125] https://docs.python.org/3.5/library/functions.html#float
[126] https://docs.python.org/3.5/library/fractions.html#fractions.Fraction
[127] https://docs.python.org/3.5/library/stdtypes.html#str
[128] https://docs.python.org/3.5/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.

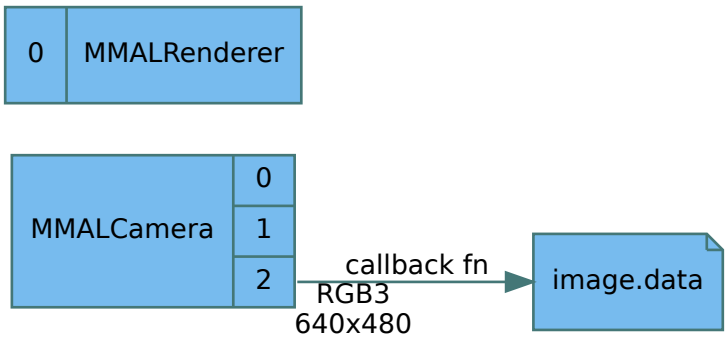
The callback will be given two parameters: the `MMALPort` responsible for producing the data, and the `MMALBuffer` containing the data. It is expected to return a `bool`[129] which will be `False` if further data is expected and `True` if no further data is expected. If `True` is returned, the callback 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:



16.1.7 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
```

(continues on next page)

[129] https://docs.python.org/3.5/library/functions.html#bool

(continued from previous page)

```
(<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:

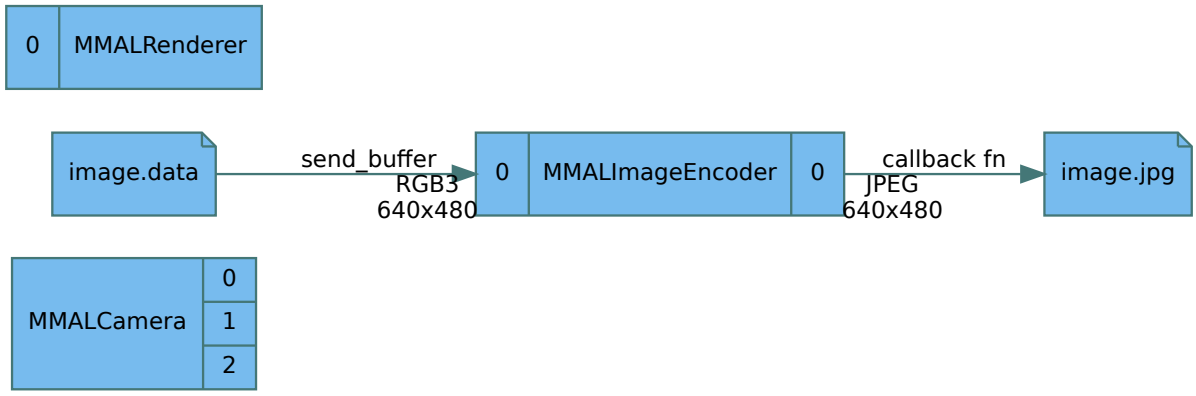
```
>>> rgb_data = io.open('image.data', 'rb')
>>> jpg_data = io.open('image.jpg', 'wb')
>>> def image_callback(port, buf):
...     jpg_data.write(buf.data)
...     return bool(buf.flags & mmal.MMAL_BUFFER_HEADER_FLAG_FRAME_END)
...
>>> encoder.outputs[0].enable(image_callback)
```

16.1.8 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(lambda 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()
```

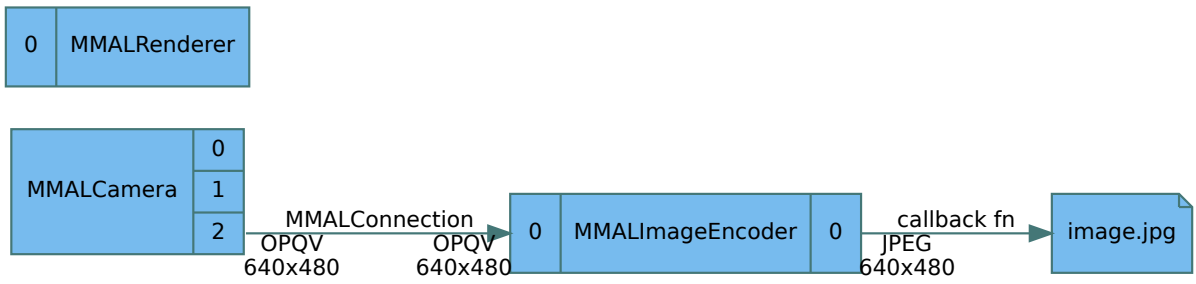
Congratulations again! You've just produced a hardware-accelerated JPEG encoding. The following illustrates the state of the system at the moment (note the camera and renderer still exist; they're just not connected to anything at the moment):



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:



16.1.9 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`[130] 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()
```

(continues on next page)

[130] https://docs.python.org/3.5/library/threading.html#module-threading

(continued from previous page)

```
...         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 picamerax operates under the hood.

The major difference between picamerax and a "typical" MMAL setup is that upon construction, the `PiCamera` class constructs both a `MMALCamera` (accessible as `PiCamera._camera`) *and* a `MMALSplitter` (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` component which is switched with a `MMALRenderer` when the preview is started.

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

16.1.10 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()`[131] outputs (in particular, it can be useful to simply evaluate a `MMALBuffer` to see what flags it's got and how much data is stored in it). Also, there's the `print_pipeline()` 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
    bitrate    0bps                      0bps         bitrate       0bps
     frame     640x480@0fps         640x480@0fps       frame       0x0@0fps
```

[131] https://docs.python.org/3.5/library/functions.html#repr

16.1.11 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 picamerax 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`, `MMALImageEncoder`, 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 `picamerax.array.PiArrayTransform`. This derives from `MMALPythonComponent` and provides the useful capability of providing the source and target buffers as numpy arrays containing RGB data:

```
>>> from picamerax 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 back-

ground 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.

16.1.12 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[132] (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` class instead, and the Pillow library[133] to perform compositing on YUV420 in the manner just described:

```
from picamerax 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
        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
```

(continues on next page)

[132] https://pillow.readthedocs.io/
[133] https://pillow.readthedocs.io/

(continued from previous page)

```
            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()` 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` 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 picamerax 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:
```

(continues on next page)

(continued from previous page)

```
            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
```

(continues on next page)

(continued from previous page)

```
        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()
```

(continues on next page)

(continued from previous page)

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

16.1.13 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).

16.2 Components

16.3 Ports

16.4 Connections

16.5 Buffers

16.6 Python Extensions

16.7 Debugging

The following functions are useful for quickly dumping the state of a given MMAL pipeline:

Note: It is also worth noting that most classes, in particular `MMALVideoPort` and `MMALBuffer` have useful `repr()` [134] outputs which can be extremely useful with simple `print()` [135] calls for debugging.

16.8 Utility Functions

The following functions are provided to ease certain common operations in the picamerax library. Users of `mmalobj` may find them handy in various situations:

[134] https://docs.python.org/3.5/library/functions.html#repr
[135] https://docs.python.org/3.5/library/functions.html#print

CHAPTER 17

Change log

17.1 Release 1.13 (2017-02-25)

1.13 includes numerous bug fixes and several major enhancements, mostly in the *mmalobj* (page 109) layer:

- 10 second captures should now work with the V2 module as the default `CAPTURE_TIMEOUT` has been increased to 60 seconds (#284[136])
- A bug in `copy_to()` caused it to copy nothing when it encountered "unknown" timestamps in the stream (#302[137], #319[138], #357[139])
- A silly typo in code used by `PiRGBArray` was fixed (#321[140])
- A bug in `capture_continuous()` which caused duplicate frames in the output was fixed (#311[141])
- Bitrate limits were removed on MJPEG, and full checking of H264 bitrates and macroblocks/s was implemented (#315[142])
- A bug was fixed in the `sensor_mode` attribute which prevented it from being set after construction (#324[143])
- A bug in the custom encoders example was fixed (#337[144])
- Fixed a rare race condition that occurred when multiple splitter ports were in use (#344[145])
- Recording overlays is now possible, but currently requires using the lower level *mmalobj* (page 109) layer (#196[146])
- Capturing YUV arrays via `PiYUVArray` is faster, thanks to GitHub user goosst (#308[147])
- Added the ability to specify a restart interval for JPEG encoding (#369[148])

[136] https://github.com/waveform80/picamerax/issues/284
[137] https://github.com/waveform80/picamerax/issues/302
[138] https://github.com/waveform80/picamerax/issues/319
[139] https://github.com/waveform80/picamerax/issues/357
[140] https://github.com/waveform80/picamerax/issues/321
[141] https://github.com/waveform80/picamerax/issues/311
[142] https://github.com/waveform80/picamerax/issues/315
[143] https://github.com/waveform80/picamerax/issues/324
[144] https://github.com/waveform80/picamerax/issues/337
[145] https://github.com/waveform80/picamerax/issues/344
[146] https://github.com/waveform80/picamerax/issues/196
[147] https://github.com/waveform80/picamerax/issues/308
[148] https://github.com/waveform80/picamerax/issues/369

- Added a property allowing users to manually specify a `framerate_range` for the camera (#374[149])
- Added support for partially transparent overlays in RGBA format (#199[150])
- Improved MJPEG web-streaming recipe, many thanks to GitHub user BigNerd95! (#375[151])

Substantial work has also gone into improving the documentation. In particular:

- The *Advanced Recipes* (page 23) 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 63) 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 109) 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

17.2 Release 1.12 (2016-07-03)

1.12 is almost entirely a bug fix release:

- Fixed issue with unencoded captures in Python 3 (#297[152])
- Fixed several Python 3 bytes/unicode issues that were related to #297[153] (I'd erroneously run the picamerax test suite twice against Python 2 instead of 2 and 3 when releasing 1.11, which is how these snuck in)
- Fixed multi-dimensional arrays for overlays under Python 3
- Finished alternate CIE constructors for the `Color` class

17.3 Release 1.11 (2016-06-19)

1.11 on the surface consists mostly of enhancements, but underneath includes a major re-write of picamerax's core:

- Direct capture to buffer-protocol objects, such as numpy arrays (#241[154])
- Add `request_key_frame()` method to permit manual request of an I-frame during H264 recording; this is now used implicitly by `split_recording()` (#257[155])
- Added `timestamp` attribute to query camera's clock (#212[156])
- Added `framerate_delta` to permit small adjustments to the camera's framerate to be performed "live" (#279[157])
- Added `clear()` and `copy_to()` methods to `PiCameraCircularIO` (#216[158])
- Prevent setting attributes on the main `PiCamera` class to ease debugging in educational settings (#240[159])

[149] https://github.com/waveform80/picamerax/issues/374
[150] https://github.com/waveform80/picamerax/issues/199
[151] https://github.com/waveform80/picamerax/issues/375
[152] https://github.com/waveform80/picamerax/issues/297
[153] https://github.com/waveform80/picamerax/issues/297
[154] https://github.com/waveform80/picamerax/issues/241
[155] https://github.com/waveform80/picamerax/issues/257
[156] https://github.com/waveform80/picamerax/issues/212
[157] https://github.com/waveform80/picamerax/pull/279
[158] https://github.com/waveform80/picamerax/issues/216
[159] https://github.com/waveform80/picamerax/issues/240

- Due to the core re-writes in this version, you may require cutting edge firmware (`sudo rpi-update`) if you are performing unencoded captures, unencoded video recording, motion estimation vector sampling, or manual sensor mode setting.
- Added property to control preview's `resolution` separately from the camera's `resolution` (required for maximum resolution previews on the V2 module - #296[160]).

There are also several bug fixes:

- Fixed basic stereoscopic operation on compute module (#218[161])
- Fixed accessing framerate as a tuple (#228[162])
- Fixed hang when invalid file format is specified (#236[163])
- Fixed multiple bayer captures with `capture_sequence()` and `capture_continuous()` (#264[164])
- Fixed usage of "falsy" custom outputs with `motion_output` (#281[165])

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!

17.4 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[166])
- A new *intra_refresh* attribute is added to the `start_recording()` method permitting control of the intra-frame refresh method (#193[167])
- 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[168])
- The new annotate V3 struct is now supported, providing custom background colors for annotations, and configurable text size. As part of this work a new `Color` class was introduced for representation and manipulation of colors (#203[169])
- Reverse enumeration of frames in `PiCameraCircularIO` is now supported efficiently (without having to convert frames to a list first) (#204[170])
- Finally, the API documentation has been re-worked as it was getting too large to comfortably load on all platforms (no ticket)

17.5 Release 1.9 (2015-01-01)

1.9 consists mostly of bug fixes with a couple of minor new features:

- The camera's sensor mode can now be forced to a particular setting upon camera initialization with the new `sensor_mode` parameter to `PiCamera` (#165[171])

[160] https://github.com/waveform80/picamerax/issues/296
[161] https://github.com/waveform80/picamerax/issues/218
[162] https://github.com/waveform80/picamerax/issues/228
[163] https://github.com/waveform80/picamerax/issues/236
[164] https://github.com/waveform80/picamerax/issues/264
[165] https://github.com/waveform80/picamerax/issues/281
[166] https://github.com/waveform80/picamerax/issues/184
[167] https://github.com/waveform80/picamerax/issues/193
[168] https://github.com/waveform80/picamerax/issues/198
[169] https://github.com/waveform80/picamerax/issues/203
[170] https://github.com/waveform80/picamerax/issues/204
[171] https://github.com/waveform80/picamerax/issues/165

- The camera's initial framerate and resolution can also be specified as keyword arguments to the `PiCamera` initializer. This is primarily intended to reduce initialization time (#180[172])
- Added the `still_stats` attribute which controls whether an extra statistics pass is made when capturing images from the still port (#166[173])
- Fixed the `led` attribute so it should now work on the Raspberry Pi model B+ (#170[174])
- 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[175]) * Fixed a long standing issue with MJPEG recording which caused camera lockups when resolutions greater than VGA were used (#47[176] and #179[177])
- Fixed a bug with incorrect frame metadata in `PiCameraCircularIO`. Unfortunately this required breaking backwards compatibility to some extent. If you use this class and rely on the frame metadata, please familiarize yourself with the new `complete` attribute (#177[178])
- Fixed a bug which caused `PiCameraCircularIO` to ignore the splitter port it was recording against (#176[179])
- Several documentation issues got fixed too (#167[180], #168[181], #171[182], #172[183], #182[184])

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!

17.6 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[185])
- 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[186])
- Text annotation functionality has been extended; up to 255 characters are now possible, and the new `annotate_frame_num` attribute adds rendering of the current frame number. In addition, the new `annotate_background` flag permits a dark background to be rendered behind all annotations for contrast (#160[187])
- Arbitrary image overlays can now be drawn on the preview using the new `add_overlay()` method. A new recipe has been included demonstrating overlays from PIL images and numpy arrays. As part of this work the preview system was substantially changed; all older scripts should continue to work but please be aware that most preview attributes are now deprecated; the new `preview` attribute replaces them (#144[188])
- Image effect parameters can now be controlled via the new `image_effect_params` attribute (#143[189])

[172] https://github.com/waveform80/picamerax/issues/180
[173] https://github.com/waveform80/picamerax/issues/166
[174] https://github.com/waveform80/picamerax/issues/170
[175] https://github.com/waveform80/picamerax/issues/174
[176] https://github.com/waveform80/picamerax/issues/47
[177] https://github.com/waveform80/picamerax/pull/179
[178] https://github.com/waveform80/picamerax/issues/177
[179] https://github.com/waveform80/picamerax/pull/176
[180] https://github.com/waveform80/picamerax/issues/167
[181] https://github.com/waveform80/picamerax/issues/168
[182] https://github.com/waveform80/picamerax/issues/171
[183] https://github.com/waveform80/picamerax/pull/172
[184] https://github.com/waveform80/picamerax/issues/182
[185] https://github.com/waveform80/picamerax/issues/149
[186] https://github.com/waveform80/picamerax/issues/153
[187] https://github.com/waveform80/picamerax/issues/160
[188] https://github.com/waveform80/picamerax/issues/144
[189] https://github.com/waveform80/picamerax/issues/143

- A bug in the handling of framerates meant that long exposures (>1s) weren't operating correctly. This *should* be fixed, but I'd be grateful if users could test this and let me know for certain (Exif metadata reports the configured exposure speed so it can't be used to determine if things are actually working) (#135[190])
- 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[191])
- Finally, the confusingly named `crop` attribute was changed to a deprecated alias for the new `zoom` attribute (#146[192])

17.7 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[193])
- Support for more than one camera on the compute module has been added, but hasn't been tested yet (#84[194])
- The `exposure_mode 'off'` has been added to allow locking down the exposure time, along with some new recipes demonstrating this capability (#116[195])
- The valid values for various attributes including `awb_mode`, `meter_mode`, and `exposure_mode` are now automatically included in the documentation (#130[196])
- Support for unencoded formats (YUV, RGB, etc.) has been added to the `start_recording()` method (#132[197])
- A couple of analysis classes have been added to *picamerax.array* (page 107) to support the new unencoded recording formats (#139[198])
- Several issues in the `PiBayerArray` class were fixed; this should now work correctly with Python 3, and the `demosaic()` method should operate correctly (#133[199], #134[200])
- A major issue with multi-resolution recordings which caused all recordings to stop prematurely was fixed (#136[201])
- Finally, an issue with the example in the documentation for custom encoders was fixed (#128[202])

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!

17.8 Release 1.6 (2014-07-21)

1.6 is half bug fixes, half new features:

- The `awb_gains` attribute is no longer write-only; you can now read it to determine the red/blue balance that the camera is using (#98[203])

[190] https://github.com/waveform80/picamerax/issues/135
[191] https://github.com/waveform80/picamerax/issues/154
[192] https://github.com/waveform80/picamerax/issues/146
[193] https://github.com/waveform80/picamerax/issues/16
[194] https://github.com/waveform80/picamerax/issues/84
[195] https://github.com/waveform80/picamerax/issues/116
[196] https://github.com/waveform80/picamerax/issues/130
[197] https://github.com/waveform80/picamerax/issues/132
[198] https://github.com/waveform80/picamerax/issues/139
[199] https://github.com/waveform80/picamerax/issues/133
[200] https://github.com/waveform80/picamerax/issues/134
[201] https://github.com/waveform80/picamerax/issues/136
[202] https://github.com/waveform80/picamerax/issues/128
[203] https://github.com/waveform80/picamerax/issues/98

- The new read-only `exposure_speed` attribute will tell you the shutter speed the camera's auto-exposure has determined, or the shutter speed you've forced with a non-zero value of `shutter_speed` (#98[204])
- The new read-only `analog_gain` and `digital_gain` attributes can be used to determine the amount of gain the camera is applying at a couple of crucial points of the image processing pipeline (#98[205])
- The new `drc_strength` attribute can be used to query and set the amount of dynamic range compression the camera will apply to its output (#110[206])
- The *intra_period* parameter for `start_recording()` can now be set to *0* (which means "produce one initial I-frame, then just P-frames") (#117[207])
- The *burst* parameter was added to the various `capture()` methods; users are strongly advised to read the cautions in the docs before relying on this parameter (#115[208])
- One of the advanced recipes in the manual ("splitting to/from a circular stream") failed under 1.5 due to a lack of splitter-port support in the circular I/O stream class. This has now been rectified by adding a *splitter_port* parameter to the constructor of `PiCameraCircularIO` (#109[209])
- Similarly, the *array extensions* (page 107) 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[210])
- A bug that caused picamerax to fail when the display was disabled has been squashed (#120[211])

As always, many thanks to the community for another great set of bug reports!

17.9 Release 1.5 (2014-06-11)

1.5 fixed several bugs and introduced a couple of major new pieces of functionality:

- The new *picamerax.array* (page 107) module provides a series of custom output classes which can be used to easily obtain numpy arrays from a variety of sources (#107[212])
- The *motion_output* parameter was added to `start_recording()` to enable output of motion vector data generated by the H.264 encoder. A couple of new recipes were added to the documentation to demonstrate this (#94[213])
- 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[214])
- An example recipe was added to the documentation covering loading and conversion of raw Bayer data (#95[215])
- 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[216])
- An issue with out-of-order calls to `stop_recording()` when multiple recordings were active was resolved (#105[217])

[204] https://github.com/waveform80/picamerax/issues/98
[205] https://github.com/waveform80/picamerax/issues/98
[206] https://github.com/waveform80/picamerax/issues/110
[207] https://github.com/waveform80/picamerax/issues/117
[208] https://github.com/waveform80/picamerax/issues/115
[209] https://github.com/waveform80/picamerax/issues/109
[210] https://github.com/waveform80/picamerax/issues/121
[211] https://github.com/waveform80/picamerax/issues/120
[212] https://github.com/waveform80/picamerax/issues/107
[213] https://github.com/waveform80/picamerax/issues/94
[214] https://github.com/waveform80/picamerax/issues/97
[215] https://github.com/waveform80/picamerax/issues/95
[216] https://github.com/waveform80/picamerax/issues/103
[217] https://github.com/waveform80/picamerax/issues/105

- Finally, picamerax 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[218])

17.10 Release 1.4 (2014-05-06)

1.4 mostly involved bug fixes with a couple of new bits of functionality:

- The *sei* parameter was added to `start_recording()` to permit inclusion of "Supplemental Enhancement Information" in the output stream (#77[219])
- The `awb_gains` attribute was added to permit manual control of the auto-white-balance red/blue gains (#74[220])
- A bug which cause `split_recording()` to fail when low framerates were configured was fixed (#87[221])
- A bug which caused picamerax to fail when used in UNIX-style daemons, unless the module was imported *after* the double-fork to background was fixed (#85[222])
- A bug which caused the `frame` attribute to fail when queried in Python 3 was fixed (#80[223])
- A bug which caused raw captures with "odd" resolutions (like 100x100) to fail was fixed (#83[224])

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[225])

Many thanks to the community for yet more excellent quality bug reports!

17.11 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[226])
- The `record_sequence()` method was added to provide a cleaner interface for recording multiple consecutive video clips (#53[227])
- The *splitter_port* parameter was added to all capture methods and `start_recording()` to permit recording multiple simultaneous video streams (presumably with different options, primarily *resize*) (#56[228])
- The limits on the `framerate` attribute were increased after firmware #656 introduced numerous new camera modes including 90fps recording (at lower resolutions) (#65[229])

And partly bug fixes:

- It was reported that Exif metadata (including thumbnails) wasn't fully recorded in JPEG output (#59[230])

[218] https://github.com/waveform80/picamerax/issues/89
[219] https://github.com/waveform80/picamerax/issues/77
[220] https://github.com/waveform80/picamerax/issues/74
[221] https://github.com/waveform80/picamerax/issues/87
[222] https://github.com/waveform80/picamerax/issues/85
[223] https://github.com/waveform80/picamerax/issues/80
[224] https://github.com/waveform80/picamerax/issues/83
[225] https://github.com/waveform80/picamerax/issues/73
[226] https://github.com/waveform80/picamerax/issues/52
[227] https://github.com/waveform80/picamerax/issues/53
[228] https://github.com/waveform80/picamerax/issues/56
[229] https://github.com/waveform80/picamerax/issues/65
[230] https://github.com/waveform80/picamerax/issues/59

- Raw captures with `capture_continuous()` and `capture_sequence()` were broken (#55[231])

17.12 Release 1.2 (2014-02-02)

1.2 was mostly a bug fix release:

- A bug introduced in 1.1 caused `split_recording()` to fail if it was preceded by a video-port-based image capture (#49[232])
- 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[233])
- 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[234])

17.13 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[235])
- An assertion error raised when performing raw captures with an active resize parameter was fixed (#46[236])
- A couple of documentation enhancements made it in (#41[237] and #47[238])

17.14 Release 1.0 (2014-01-11)

In 1.0 the major features added were:

- Debian packaging! (#12[239])
- The new `frame` attribute permits querying information about the frame last written to the output stream (number, timestamp, size, keyframe, etc.) (#34[240], #36[241])
- All capture methods (`capture()` et al), and the `start_recording()` method now accept a `resize` parameter which invokes a resizer prior to the encoding step (#21[242])
- A new `PiCameraCircularIO` stream class is provided to permit holding the last *n* seconds of video in memory, ready for writing out to disk (or whatever you like) (#39[243])
- There's a new way to specify raw captures - simply use the format you require with the capture method of your choice. As a result of this, the `raw_format` attribute is now deprecated (#32[244])

Some bugs were also fixed:

[231] https://github.com/waveform80/picamerax/issues/55
[232] https://github.com/waveform80/picamerax/issues/49
[233] https://github.com/waveform80/picamerax/issues/23
[234] https://github.com/waveform80/picamerax/issues/48
[235] https://github.com/waveform80/picamerax/issues/40
[236] https://github.com/waveform80/picamerax/issues/46
[237] https://github.com/waveform80/picamerax/pull/41
[238] https://github.com/waveform80/picamerax/issues/47
[239] https://github.com/waveform80/picamerax/issues/12
[240] https://github.com/waveform80/picamerax/issues/34
[241] https://github.com/waveform80/picamerax/issues/36
[242] https://github.com/waveform80/picamerax/issues/21
[243] https://github.com/waveform80/picamerax/issues/39
[244] https://github.com/waveform80/picamerax/issues/32

- GPIO.cleanup is no longer called on `close()` (#35[245]), and GPIO set up is only done on first use of the `led` attribute which should resolve issues that users have been having with using picamerax in conjunction with GPIO
- Raw RGB video-port based image captures are now working again too (#32[246])

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

- The continuous method was removed; this was replaced by `capture_continuous()` in 0.5 (#7[247])

17.15 Release 0.8 (2013-12-09)

In 0.8 the major features added were:

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

In addition a few bugs were fixed:

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

17.16 Release 0.7 (2013-11-14)

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

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

17.17 Release 0.6 (2013-10-30)

In 0.6, the major features added were:

- New `'raw'` format added to all capture methods (`capture()`, `capture_continuous()`, and `capture_sequence()`) to permit capturing of raw sensor data
- New `raw_format` attribute to permit control of raw format (defaults to `'yuv'`, only other setting currently is `'rgb'`)

[245] https://github.com/waveform80/picamerax/issues/35
[246] https://github.com/waveform80/picamerax/issues/32
[247] https://github.com/waveform80/picamerax/issues/7

- New `shutter_speed` attribute to permit manual control of shutter speed (defaults to 0 for automatic shutter speed, and requires latest firmware to operate - use `sudo rpi-update` to upgrade)
- New "Recipes" chapter in the documentation which demonstrates a wide variety of capture techniques ranging from trivial to complex

17.18 Release 0.5 (2013-10-21)

In 0.5, the major features added were:

- New `capture_sequence()` method
- `continuous()` method renamed to `capture_continuous()`. Old method name retained for compatiblity until 1.0.
- *use_video_port* option for `capture_sequence()` and `capture_continuous()` to allow rapid capture of JPEGs via video port
- New `framerate` attribute to control video and rapid-image capture frame rates
- Default value for `ISO` changed from 400 to 0 (auto) which fixes `exposure_mode` not working by default
- *intraperiod* and *profile* options for `start_recording()`

In addition a few bugs were fixed:

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

Many thanks to the community for the bug reports!

17.19 Release 0.4 (2013-10-11)

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

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

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

17.20 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.

17.21 Release 0.2

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

CHAPTER 18

License

The *bayer pattern diagram* (page 46) in the documentation is derived from Bayer_pattern_on_sensor.svg[249] which is copyright (c) Colin Burnett (User:Cburnett) on Wikipedia, modified under the terms of the GPL:

This work is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or any later version. This work is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. See version 2 and version 3 of the GNU General Public License for more details.

---

[248] dave@waveform.org.uk
[249] https://en.wikipedia.org/wiki/File:Bayer_pattern_on_sensor.svg

The *YUV420 planar diagram* (page 24) in the documentation is Yuv420.svg[250] created by Geoff Richards (User:Qef) on Wikipedia, released into the public domain.

[250] https://en.wikipedia.org/wiki/File:Yuv420.svg

Python Module Index

p

Index

P