The Picamera2 Library

A libcamera-based Python library for Raspberry Pi cameras

Raspberry Pi Ltd

Colophon

© 2022 Raspberry Pi Ltd

This documentation is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND).

build-date: 2022-08-19
build-version: 26d3737-clean

Legal Disclaimer Notice

Table of Contents

Colophon . . . 1
Legal Disclaimer Notice . . . 1
1. Introduction . . . 4
2. Getting Started . . . 5
2.1. Requirements . . . 5
2.2. Installation and Updating . . . 5
2.3. A First Example . . . 6
2.4. *Picamera2*'s high level API . . . 7
2.5. Additional Software . . . 7
2.5.1. OpenCV . . . 7
2.5.2. TensorFlow Lite . . . 7
2.5.3. FFmpeg . . . 8
2.6. Further Examples . . . 8
3. Preview Windows . . . 9
3.1. Preview Window Parameters . . . 9
3.2. Preview Window Implementations . . . 9
3.2.1. QTGL Preview . . . 9
3.2.2. DRM/KMS Preview . . . 10
3.2.3. QT Preview . . . 10
3.2.4. NULL Preview . . . 11
3.3. Starting and Stopping Previews . . . 11
3.4. Remote Preview Windows . . . 12
3.5. Further Preview Topics . . . 12
3.6. Further Examples . . . 13
4. Configuring the Camera . . . 14
4.1. Generating and Using a Camera Configuration . . . 14
4.2. Configurations in more Detail . . . 14
4.2.1. General Configuration Parameters . . . 15
4.2.2. Stream Configuration Parameters . . . 18
4.2.3. Configurations and Runtime Camera Controls . . . 20
4.3. Configuration Objects . . . 21
4.4. Further Examples . . . 23
5. Camera Controls and Properties . . . 24
5.1. Camera Controls . . . 24
5.1.1. How to set Camera Controls . . . 24
5.1.2. Object Syntax for Camera Controls . . . 25
5.2. Camera Properties . . . 26
5.3. Further Examples . . . 26
6. Capturing Images and Requests . . . 28
6.1. Capturing Images . . . 28
6.1.1. Capturing Arrays . . . 28
6.1.2. Capturing *PIL* Images . . . 29
6.1.3. Switching Camera Mode and capturing . . . 29
6.1.4. Capturing straight to Files and file-like Objects . . . 30
6.2. Capturing Metadata . . . 30
6.3. Capturing Multiple Images at Once . . . 31
6.4. Capturing Requests . . . 32
6.5. Asynchronous Capture . . . 33
6.6. High Level Capture API . . . 35
6.6.1. start_and_capture_file . . . 35
6.6.2. start_and_capture_files . . . 35
6.7. Further Examples . . . 36
7. Capturing Videos . . . 37
7.1. Encoders . . . 37
7.1.1. H264Encoder . . . 38

7.1.2. JpegEncoder . . . 38
7.1.3. MJPEGEncoder . . . 39
7.2. Outputs . . . 39
7.2.1. FileOutput . . . 39
7.2.2. FfmpegOutput . . . 40
7.2.3. CircularOutput . . . 41
7.3. High Level Video Recording API . . . 41
7.4. Further Examples . . . 42
8. Advanced Topics . . . 43
8.1. Display Overlays . . . 43
8.2. The Event Loop . . . 44
8.2.1. Using the Event Loop Callbacks . . . 44
8.2.2. Dispatching Tasks into the Event Loop . . . 45
8.3. Pixel Formats and Memory Considerations . . . 47
8.4. Buffer Allocations and Queues . . . 48
8.5. Using the Camera in Qt Applications . . . 49
9. Application Notes . . . 53
9.1. Streaming to a Network . . . 53
9.2. Output using FFmpeg . . . 53
9.2.1. HLS Live Stream . . . 53
9.2.2. MPEG-DASH Live Stream . . . 53
9.2.3. Sending an MPEG-2 Transport Stream to a Socket . . . 54
9.3. Multiple Outputs . . . 54
9.4. Manipulate Camera Buffers in Place . . . 55
Appendix A: Pixel and Image Formats . . . 56
Appendix B: Camera Configuration Parameters . . . 58
Appendix C: Camera Controls . . . 60
Appendix D: Camera Properties . . . 61

Chapter 1. Introduction

Picamera2 is a Python library that gives convenient access to the camera system of the Raspberry Pi. It is designed for cameras connected with the flat ribbon cable directly to the connector on the Raspberry Pi itself, and does not support any other kinds of camera, such as those connected via USB or over networks.

Figure 1. A Raspberry Pi with a supported camera

Picamera2 is built on top of the open source *libcamera* project, which provides support for complex camera systems in Linux. In fact, *Picamera2* directly uses the Python bindings supplied by *libcamera*, although the *Picamera2* API provides access at a higher level. Most users will find it significantly easier to use for Raspberry Pi applications than *libcamera*'s own bindings, and *Picamera2* is tuned specifically to address the capabilities of the Raspberry Pi's built-in camera and imaging hardware.

For those familiar with it, *Picamera2* is the replacement for the legacy *PiCamera* Python library. It provides broadly the same facilities, although, by running through the *libcamera* stack, many of these capabilities are exposed differently. *Picamera2* actually provides a very direct and more accurate view of the Pi's camera system, and makes it easy for Python applications to make use of them.

Those still using *Raspberry Pi OS Legacy*, or using the legacy camera stack, should continue to use the old *PiCamera* library.

NOTE

This document assumes general familiarity with Raspberry Pis and Python programming. A working understanding of images and how they can be represented as a 2-dimensional array of pixels will also be highly beneficial. For a deeper understanding of *Picamera2*, some basic knowledge of Python's *numpy* library will be helpful. For some more advanced use-cases, an awareness of *OpenCV* (the Python *cv2* module) will also be useful.

Software Version

This manual describes *Picamera2* version 0.3.1 which is at the time of writing the most up to date release.

Chapter 2. Getting Started

2.1. Requirements

Picamera2 is designed for systems running either Raspberry Pi OS version or Raspberry Pi OS Lite, version *Bullseye* or later. It is pre-installed in current images downloaded from the Raspberry Pi website, or installed via the Raspberry Pi Imager tool. Users with older images should consider updating them or proceed to the installation instructions.

Picamera2 can operate in a headless manner, not requiring an attached screen or keyboard. However, when first setting up such a system we would recommend attaching a keyboard and screen if possible as it can make trouble-shooting easier.

Raspberry Pi OS *Bullseye* (or later) by default runs the *libcamera* camera stack, which is required for *Picamera2*. You can check that *libcamera* is working by opening a command window and typing:

```
libcamera-hello
```

You should see a camera preview window for about 5 seconds. If you do not, please refer to the Raspberry Pi camera documentation.

Using lower-powered devices

Some lower-powered devices, such as the Raspberry Pi Zero, are generally much slower at running software like X-Windows. Correspondingly, performance may be poor trying to run the camera system with a preview window that has to display images through the X-Windows display stack.

On such devices we would recommend either not displaying the images, or displaying them without X-Windows. The Pi can be configured to boot to the console (and not X-Windows) using the `raspi-config` tool, or if you are running X-Windows it can temporarily be suspended by holding the `Ctrl+Alt+F1` keys (and use `Ctrl+Alt+F7` to return again).

2.2. Installation and Updating

Picamera2 can be installed with or without all the X-Windows and GUI dependencies. Installation *without* these dependendencies would normally be recommended for Raspberry Pi OS Lite users.

To install or update the *Picamera2* including all the X-Windows and GUI dependencies, users should enter:

```
sudo apt install -y python3-libcamera python3-kms++
sudo apt install -y python3-pyqt5 python3-opengl python3-prctl libatlas-base-dev ffmpeg
python3-pip
pip3 install numpy --upgrade
sudo apt install -y python3-picamera2
```

Users who do not want the X-Windows or GUI dependencies should instead use:

```
sudo apt install -y python3-libcamera python3-kms++
sudo apt install -y python3-prctl libatlas-base-dev ffmpeg libopenjp2-7 python3-pip
pip3 install numpy --upgrade
sudo apt install -y python3-picamera2 --no-install-recommends
```

Configuring the `/boot/config.txt` file

Normally no changes should be required to the `/boot/config.txt` file from the one supplied when the operating system was installed.

Some users, for some applications, may find themselves needing to allocate more memory to the camera system. In this case, please consider increasing the amount of available CMA memory.

In the past, legacy camera stack users may have increased the amount of `gpu_mem` to enable the old camera system to run. *Picamera2* does not use this type of memory, so any such lines should be deleted from your `/boot/config.txt` file as they will simply cause system memory to be wasted.

2.3. A First Example

The following script will:

1. Open the camera system.
2. Generate a camera configuration suitable for preview.
3. Configure the camera system with that preview configuration.
4. Start the preview window.
5. Start the camera running.
6. Wait for 2 seconds and capture a JPEG file (still in the preview resolution).

NOTE

Users of Pi 3 or earlier devices will need to enable *Glamor* in order for this example script using X-Windows to work. To do this, run `sudo raspi-config` in a command window, choose *Advanced Options* and then enable *Glamor* graphic acceleration. Finally reboot your device.

X-Windows users should enter:

```python
from picamera2 import Picamera2, Preview
import time

picam2 = Picamera2()
camera_config = picam2.create_preview_configuration()
picam2.configure(camera_config)
picam2.start_preview(Preview.QTGL)
picam2.start()
time.sleep(2)
picam2.capture_file("test.jpg")
```

Non X-Windows users should use the same script, but replacing `Preview.QTGL` by `Preview.DRM`, so as to use the non X-Windows preview implementation:

```python
from picamera2 import Picamera2, Preview
import time

picam2 = Picamera2()
camera_config = picam2.create_preview_configuration()
picam2.configure(camera_config)
picam2.start_preview(Preview.DRM)
picam2.start()
time.sleep(2)
```

```
picam2.capture_file("test.jpg")
```

2.4. *Picamera2*'s high level API

Picamera2 has some high level and very convenient functions for capturing images and video recordings. These are ideal for those who do not want to know too much about the details of how *Picamera2* works, though more advanced users will want a deeper understanding of what happens under the hood, as we saw above.

If you simply want to capture an image, the following is sufficient:

```
from picamera2 import Picamera2

picam2 = Picamera2()

picam2.start_and_capture_file("test.jpg")
```

You can capture multiple images with the `start_and_capture_files` function. Or, to record a 5 second video:

```
from picamera2 import Picamera2

picam2 = Picamera2()

picam2.start_and_record_video("test.mp4", duration=5)
```

We will learn more about these functions later on, for both still images and video.

2.5. Additional Software

When using *Picamera2*, it's often useful to install other packages. For convenience we list some common ones here.

2.5.1. OpenCV

OpenCV is not a requirement for *Picamera2*, though a number of examples use it. It can by installed from `apt` very easily as follows, avoiding the long build times involved in some other methods:

```
sudo apt install -y python3-opencv
sudo apt install -y opencv-data
```

2.5.2. TensorFlow Lite

TensorFlow Lite can be installed with:

```
pip3 install tflite-runtime
```

2.5.3. FFmpeg

Some features of *Picamera2* make use of the *FFmpeg* library. Normally this should be installed by default on a Raspberry Pi, but in case it isn't the following should fetch it:

```
sudo apt install -y ffmpeg
```

2.6. Further Examples

Throughout this guide we'll give lots of examples, but we'll also highlight some of the ones in *Picamera2*'s GitHub repository.

The `examples` folder in the repository can be found here. There are some additional examples framed as *Qt* applications which can be found here.

Chapter 3. Preview Windows

3.1. Preview Window Parameters

In the previous section we have already seen two different types of preview window. There are in fact four different versions, which we shall discuss below.

All four preview implementations accept exactly the same parameters so that they are interchangeable:

- `x` - the x-offset of the preview window
- `y` - the y-offset of the preview window
- `width` - the width of the preview window
- `height` - the height of the preview window
- `transform` - a transform that allows the camera image to be horizontally and/or vertically flipped on the display.

All the parameters are optional and default values will be chosen if omitted. The following example will place an 800x600 pixel preview window at (100, 200) on the display, and will horizontally mirror the camera preview image:

```
from picamera2 import Picamera2, Preview
from libcamera import Transform

picam2 = Picamera2()
picam2.start_preview(Preview.QTGL, x=100, y=200, width=800, height=600,
transform=Transform(hflip=1))
picam2.start()
```

The supported transforms are:

- `Transform()` - the identity transform, which is the default
- `Transform(hflip=1)` - horizontal flip
- `Transform(vflip=1)` - vertical flip
- `Transform(hflip=1, vflip=1)` - horizontal and vertical flip (equivalent to a 180 degree rotation).

It's important to realise that the *display transform* discussed here does not have any effect on the actual images received from the camera. It only applies the requested transform as it renders the pixels onto the screen. We'll encounter camera transforms again when it comes actually to transforming the images as the camera delivers them.

Please also note that in the example above, the `start_preview()` function must be called before the call to `picam2.start()`.

Finally, if the camera images have a different aspect ratio to the preview window, they will be letter- or pillar-boxed to fit, preserving the image's proper aspect ratio.

3.2. Preview Window Implementations

3.2.1. QTGL Preview

This preview window is implemented using the *Qt* GUI toolkit and uses *GLES* hardware graphics acceleration. It is the most efficient way to display images on the screen when using X-Windows and we would recommend it in nearly all circumstances when X-Windows is required.

The QTGL preview window can be started with:

```
from picamera2 import Picamera2, Preview

picam2 = Picamera2()
picam2.start_preview(Preview.QTGL)
```

The QTGL preview window is not recommended when the image needs to be shown on a remote display (not connected to the Pi). Please refer to the QT preview window below.

Users of Pi 3 or earlier devices will need to enable *Glamor* graphic acceleration to use the QTGL preview window.

NOTE

There is a limit to the size of image that the 3D graphics hardware on the Pi can handle. For Pi 4s this limit is 4096 pixels in either dimension. For Pi 3 and earlier devices this limit is 2048 pixels. If you try to feed a larger image to the QTGL preview window it will report an error and the program will terminate.

3.2.2. DRM/KMS Preview

The DRM/KMS preview window is for use when X-Windows is not running and camera system can lease a "layer" on the display for displaying graphics. It can be started with:

```
from picamera2 import Picamera2, Preview

picam2 = Picamera2()
picam2.start_preview(Preview.DRM)
```

Because X-Windows is not running, it is not possible to grab and move or resize this window.

The DRM/KMS preview will be the natural choice for Raspberry Pi OS Lite users. It is also strongly recommended for lower powered Raspberry Pis that would find it expensive to pass a preview (for example at 30 frames per second) through the X-Windows display stack.

3.2.3. QT Preview

Like the QTGL preview, this window is also implemented using the *Qt* framework, but this time using software rendering rather than 3D hardware acceleration. As such, it is very much more costly and should always be avoided when possible. Even a Raspberry Pi 4 will start to struggle once the preview window size increases.

The QT preview can be started with:

```
from picamera2 import Picamera2, Preview

picam2 = Picamera2()
picam2.start_preview(Preview.QT)
```

The main use cases for the QT preview would include displaying the preview window onto another networked computer using X forwarding, or using the VNC remote desktop software. Under these conditions the 3D hardware accelerated implementation either does not work at all, or does not work very well.

Users of Pi 3 or earlier devices will need to enable *Glamor* graphic acceleration to use the QT preview window.

3.2.4. NULL Preview

Normally it is the preview window that actually drives the libcamera system by receiving camera images, passing them to the application, and then recycling those buffers back to libcamera once the user no longer needs them. The consequence is then that even when no preview images are being displayed, *something* still has to run in order to receive and then return those camera images.

This is exactly what the *NULL preview* does. It displays nothing; it merely drives the camera system.

The NULL preview is in fact **started automatically** whenever the camera system is started (`picam2.start()`) if no preview is yet running, which is why alternative preview windows must be started earlier. You could start the NULL preview explicitly like this:

```
from picamera2 import Picamera2, Preview

picam2 = Picamera2()
picam2.start_preview(Preview.NULL)
```

though in fact the call to `start_preview` is redundant for the NULL preview and can be omitted.

The NULL preview accepts the same parameters as the other preview implementations, but ignores them completely.

3.3. Starting and Stopping Previews

We have seen how to start preview windows, including the NULL preview which actually displays nothing. In fact, the first parameter to the `start_preview` function can take the following values:

- `None` - No preview of any kind is started. The application would have to supply its own code to drive the camera system.
- `False` - The NULL preview is started.
- `True` - One of the 3 other previews is started. *Picamera2* will attempt to auto-detect which one it should start, though this is purely on a "best efforts" basis.
- Any of the four `Preview` enum values.

Preview windows can actually be stopped after which an alternative one should be started. Generally we do not recommend starting and stopping preview windows because it can be quite expensive to open and close windows, during which time camera frames are likely to be dropped.

Furthermore, the `Picamera2.start` function accepts a `show_preview` parameter which can take on any one of these same values. This is just a convenient shorthand that allows the amount of boilerplate code to be reduced. But note that stopping the camera (`Picamera2.stop`) does not stop the preview window, so the `stop_preview` function would have to be called explicitly before starting another.

For example, the following script would start the camera system running, run for a short while, and then attempt to auto-detect which preview window to use in order actually to start displaying the images:

```
from picamera2 import Picamera2, Preview
import time

picam2 = Picamera2()
config = picam2.create_preview_configuration()
picam2.configure(config)
picam2.start()
```

```
time.sleep(2)

picam2.stop_preview()
picam2.start_preview(True)

time.sleep(2)
```

In this example:

1. The NULL preview will start automatically with the `picam2.start()` call and will run for 2 seconds.
2. It will then be stopped and a preview that displays an actual window will be started.
3. This will then run for 2 more seconds.

It's worth noting that nothing particularly bad happens if you stop the preview and then fail to restart another, or do not start another immediately. *libcamera* will simply fill up all the buffers that are available to it with camera images. But with no preview running, nothing will read out these images and recycle the buffers, so *libcamera* will simply stall. When a preview is restarted, normal operation will resume, starting with those slightly old camera images that are still queued up waiting to be read out.

NOTE

Many programmers will be familiar with the notion of an *event loop*. Each type of preview window implements an event loop to dequeue frames from the camera, so the NULL preview performs this function when no other event loop (such as the one provided by *Qt*) is running.

3.4. Remote Preview Windows

The preview window can be displayed on a remote display, for example when you have logged in to your Pi over *ssh* or through *VNC*. When using *ssh*:

- You should use the `-X` parameter (or equivalent) to set up X-forwarding.
- The QTGL (hardware accelerated) preview will not work and will result in an error message. The QT preview must be used instead, though being software rendered (and presumably travelling over a network), framerates can be expected to be significantly poorer.

When using *VNC*:

- The QTGL (hardware accelerated) window works adequately *if* you also have a display connected directly to the Pi.
- If you do not have a display connected directly to the Pi, the QTGL preview will work very poorly, and the QT preview window should be used instead.

If you are not running, or have suspended, X-Windows on the Pi but still have a display attached, you can log into the Pi without X-forwarding and use the DRM/KMS preview implementation. This will obviously appear on the display that is attached directly to the Pi.

3.5. Further Preview Topics

Further preview features are discussed in the Advanced Topics section.

- Overlays (transparent images overlaid on the live camera feed) are discussed among the Advanced Topics.
- For *Qt* applications, displaying a preview window doesn't make sense as the *Qt* framework will run the event loop. However, the underlying widgets *are* still useful and are discussed further here.

3.6. Further Examples

Most of the GitHub examples will create and start preview windows of some kind, for example:

- preview.py - starts a QTGL preview window.
- preview_drm.py - starts a DRM preview window.
- preview_x_forwarding.py - starts a QT preview window.

Chapter 4. Configuring the Camera

4.1. Generating and Using a Camera Configuration

Once a `Picamera2` object has been created, the general pattern is that a *configuration* must be generated for the camera, that configuration must be applied to the camera system (using the `Picamera2.configure` method), and then the camera system can be started.

Picamera2 provides a number of *configuration-generating methods* that can be used to provide suitable configurations for common use cases:

- `Picamera2.create_preview_configuration` will generate a configuration suitable for displaying camera preview images on the display, or prior to capturing a still image.
- `Picamera2.create_still_configuration` will generate a configuration suitable for capturing a high resolution still image.
- `Picamera2.create_video_configuration` will generate a configuration suitable for recording video files.

So for example, to set up the camera to start delivering a stream of preview images you might use:

```
from picamera2 import Picamera

picam2 = Picamera2()
config = picam2.create_preview_configuration()
picam2.configure(config)
picam2.start()
```

This is fairly typical, though the configuration-generating methods allow numerous optional parameters to adjust the resulting configuration precisely for different situations. Additionaly, once a configuration object has been created, applications are free to alter its recommendations before calling `picam2.configure`.

One thing we shall learn is that configurations are just Python dictionaries, and it's easy for us to inspect them and see what they are saying.

4.2. Configurations in more Detail

Picamera2 is easier to configure with some basic knowledge of the camera system on the Raspberry Pi. This will make it clearer why particular image streams are available and why they have some of the features and properties that they do.

The diagram below shows how the camera hardware on the Pi works.

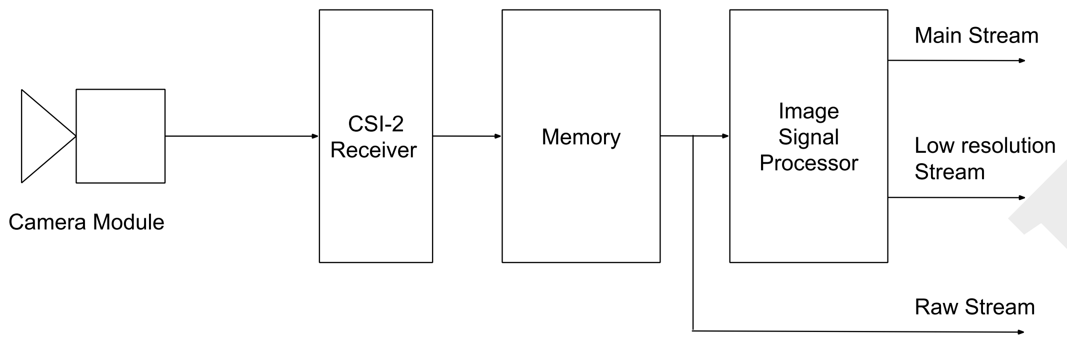

Figure 2. The Raspberry Pi's camera system

The sequence of events is as follows:

1. On the left we have the camera module which delivers images through the flat ribbon cable to the Pi. The images delivered by the camera are not human-viewable images, but need lots of work to clean them up and produce a realistic picture.
2. A hardware block called a *CSI-2 Receiver* on the Pi transfers the incoming camera image into memory.
3. The Pi has an *Image Signal Processor* (an *ISP*) which reads this image from memory. It performs all these cleaning and processing steps on the pixels that were received from the camera.
4. The ISP can produce up to *two* output images for every input frame from the camera. We designate one of them as the *main* image, and it can be in either RGB or YUV formats.
5. The second image is a lower resolution image, referred to often as the *lores* image and it must be in a YUV format. It must also be *no larger* than the *main* image.
6. Finally, the image data that was received from the sensor and written directly to memory can also be delivered to applications. This is called the *raw* image.

The configuration of Picamera2 therefore divides into:

- General parameters that apply globally to the *Picamera2* system and across the whole of the ISP.
- And per-stream configuration within the ISP that determines the output formats and sizes of the *main* and *lores* streams. We note that **the *main* stream is always defined and delivered to the application**, using default values if the application did not explicitly request one.
- Furthermore, the *raw* stream is rather different from the other streams in that it affects the mode in which the image sensor is driven more directly, so there is some separate discussion of this.
- Mostly, a configuration does not include camera settings that can be changed at runtime (such as brightness or contrast). However, certain use cases do sometimes have particular preferences about certain of these control values, and they can be stored as part of a configuration so that applying the configuration will apply the runtime controls automatically too.

4.2.1. General Configuration Parameters

The configuration parameters that affect all the output streams are:

- **transform** - whether camera images are horizontally or vertically mirrored, or both (giving a 180 degree rotation). All three streams (if present) share the same transform.
- **colour_space** - the *colour space* of the output images. The *main* and *lores* streams must always share the same colour space. The *raw* stream is always in a camera-specific colour space.
- **buffer_count** - the number of sets of buffers to allocate for the camera system. A single *set* of buffers represents one buffer for each of the streams that have been requested.

- **display** - this names which (if any) of the streams are to be shown in the preview window. It does not actually affect the camera images in any way, only what *Picamera2* does with them.
- **encode** - this names which (if any) of the streams are to be encoded if a video recording is started. This too does not affect the camera images in any way, only what *Picamera2* does with them.

4.2.1.1. More on Transforms

Transforms can be constructed as shown below:

```
>>> from libcamera import Transform

>>> Transform()
<libcamera.Transform 'identity'>
>>> Transform(hflip=True)
<libcamera.Transform 'hflip'>
>>> Transform(vflip=True)
<libcamera.Transform 'vflip'>
>>> Transform(hflip=True, vflip=True)
<libcamera.Transform 'hvflip'>
```

Transforms can be passed to all the configuration-generating methods using the `transform` keyword parameter. For example:

```
from picamera2 import Picamera2
from libcamera import Transform

picam2 = Picamera2()
preview_config = picam2.create_preview_configuration(transform=libcamera.Transform(hflip=True))
```

Picamera2 only supports the 4 transforms shown above. Other transforms (involving image transposition) exist but are not supported. If unspecified, the transform always defaults to the identity transform.

4.2.1.2. More on Colour Spaces

libcamera's implementation of colour spaces follows that of the Linux V4L2 API quite closely. Specific choices are provided for each of colour primaries, the transfer function, the YCbCr encoding matrix and the quantisation (or range). Readers are referred to the referenced document for more information.

In addition, *libcamera* provides convenient shorthand forms for commonly used colour spaces:

```
>>> from libcamera import ColorSpace
>>> ColorSpace.Jpeg()
<libcamera.ColorSpace 'JPEG'>
>>> ColorSpace.Rec709()
<libcamera.ColorSpace 'Rec709'>
```

These are in fact the only colour spaces supported by the Pi's camera system. The required choice can be passed to all the configuration-generating methods using the `colour_space` keyword parameter:

```
from picamera2 import Picamera2
from libcamera import ColorSpace
```

```
picam2 = Picamera2()
preview_config = picam2.create_preview_configuration(colour_space=ColorSpace.Jpeg())
```

When omitted, *Picamera2* will choose a default according to the use case:

- `create_preview_configuration` and `create_still_configuration` will use the JPEG colour space by default (by which we mean sRGB primaries and transfer function and full-range BT.601 YCbCr encoding).
- `create_video_configuration` will choose JPEG if the *main* stream is requesting an RGB format. For YUV formats it will choose SMPTE 170M if the resolution is less than 1280x720, otherwise Rec.709.

4.2.1.3. More on the buffer_count

This number defines how many *sets* of buffers (one for each requested stream) are allocated for the camera system to use. Allocating more buffers can mean that the camera will run more smoothly and drop fewer frames, though the downside is that, particularly at high resolutions, there may not be enough memory available.

The configuration-generating methods all choose an appropriate number of buffers for their use cases:

- `create_preview_configuration` requests 4 sets of buffers
- `create_still_configuration` requests just one set of buffers (as these are normally large full resolution buffers)
- `create_video_configuration` requests 6 buffers, as the extra work involved in encoding and outputting the video streams makes it more susceptible to jitter or delays, which is alleviated by the longer queue of buffers.

The number of buffers can be overridden in all the configuration-generating methods using the `buffer_count` keyword parameter:

```
from picamera2 import Picamera2

picam2 = Picamera2()
preview_config = picam2.create_still_configuration(buffer_count=2)
```

4.2.1.4. More on the display stream

Normally we would display the *main* stream in the preview window. In some circumstances it may be preferable to display a lower resolution image (from the *lores* stream) instead. We could use:

```
from picamera2 import Picamera2

picam2 = Picamera2()
config = picam2.create_still_configuration(lores={"size": (320, 240)}, display="lores")
```

This would reqest a full resolution *main* stream, but then also a QVGA *lores* stream which would be displayed (recall that the *main* stream is always defined even when the application does not explicitly request it).

The display parameter may take the value `None` which means that no images will be rendered to the preview window. In fact this is the default choice of the `create_still_configuration` method.

4.2.1.5. More on the encode stream

This is similar to the *display* parameter, in that it names the stream (*main* or *lores*) that will be encoded if a video recording is started. By default we would normally encode the *main* stream, but a user might have an application where they want to record a low resolution video stream instead:

```
from picamera2 import Picamera2

picam2 = Picamera2()
config = picam2.create_video_configuration(main={"size": (2048, 1536)}, lores={"size": (320,
240)}, encode="lores")
```

This would enable a QVGA stream to be recorded, while allowing 2048x1536 still images to be captured simultaneously.

The `encode` parameter may also take the value `None`, which is again the default choice of the `create_still_configuration` method.

4.2.2. Stream Configuration Parameters

All the three streams can be requested from the configuration-generating methods using the `main`, `lores` and `raw` keyword parameters, though as we have noted, a *main* stream will always be defined. The *main* stream is also the first argument to these functions, so the `main=` syntax can normally be omitted. By default the *lores* and *raw* streams are not delivered to applications.

To request one of these streams, a dictionary should be supplied. The dictionary may be an empty dictionary, at which point that stream will be generated for the application but populated completely by default values:

```
from picamera2 import Picamera2

picam2 = Picamera2()
config = picam2.create_preview_configuration(lores={})
```

Here, the *main* stream will be produced as usual, but a *lores* stream will be produced as well. By default it will have the same resolution as the *main* stream, but using the YUV420 image format.

The keys that may be specified are:

- "size" - a tuple of two values giving the width and height of the image
- "format" - a string defining one of *libcamera*'s allowable image formats.

Image Sizes

The configuration-generating functions can make minimal changes to the configuration where they detect something is invalid. But they will attempt to observe whatever values they are given even where others may be more efficient. The most obvious case of this is in relation to the image sizes.

Here, hardware restrictions means that images can be processed more efficiently if they are particular sizes. Other sizes will have to be copied more frequently in the Python world, but these special image alignment rules are somewhat arcane. They are covered in detail in the appendices.

If a user wants to request these optimal image sizes, they should use the `align_configuration` method. For example:

```
>>> from picamera2 import Picamera2
>>> picam2 = Picamera2()
>>> config = picam2.create_preview_configuration({"size": (808, 606)})
>>> config["main"]
{'format': 'XBGR8888', 'size': (808, 606)}
>>> picam2.align_configuration(config)
# Picamera2 has decided an 800x606 image will be more efficient.
>>> config["main"]
{'format': 'XBGR8888', 'size': (800, 606)}
```

```
>>> picam2.configure(config)
{'format': 'XBGR8888', 'size': (800, 606), 'stride': 3200, 'framesize': 1939200}
```

At the end we have an 800x606 image, which will result in less data copying. Observe also how, once a configuration has been applied, *Picamera2* knows some extra things: the length of each row of the image in bytes (the *stride*), and the total amount of memory every such image will occupy.

Image Formats

libcamera supports a wide variety of image formats as described in the appendices. For our purposes, however, these are some of the most common ones. For the *main* stream:

- `XBGR8888` - every pixel is packed into 32-bits, with a dummy 255 value at the end, so a pixel would look like [R, G, B, 255] when captured in Python. (Note that these format descriptions can seem counter-intuitive, but the underlying infrastructure tends to take machine *endianness* into account which can mix things up!)
- `XRGB8888` - as above, with a pixel looking like [B, G, R, 255].
- `RGB888` - 24 bits per pixel, ordered [B, G, R].
- `BGR888` - as above, but ordered [R, G, B].
- `YUV420` - YUV images with a plane of Y values followed by a quarter plane of U values and then a quarter plane of V values.

For the *lores* stream, only 'YUV420' is really used.

Raw Streams

As we have said previously, these are a bit different. We can specify the format (that includes the bit depth) and size, but only a limited selection is normally available from the sensor. They cannot generally be scaled to relatively arbitrary sizes (as the *main* and *lores* streams can, by virtue of the ISP).

Picamera2 will attempt to choose the best match for whatever you supply, but you can check for yourself what is actually available by querying the `sensor_modes` property of the *Picamera2* object. The first time you request this property it will reconfigure the sensor (several times) to determine all the sensor modes, so it is usually something you may want to do as soon as you open the camera.

For the HQ camera we obtain:

```
>>> from pprint import *
>>> from picamera2 import Picamera2
>>> picam2 = Picamera2()
# output omitted
>>> pprint(picam2.sensor_modes)
# output omitted
[{'bit_depth': 10,
  'crop_limits': (696, 528, 2664, 1980),
  'exposure_limits': (31, 66512892),
  'format': SRGGB10_CSI2P,
  'fps': 120.05,
  'size': (1332, 990),
  'unpacked': 'SRGGB10'},
 {'bit_depth': 12,
  'crop_limits': (0, 440, 4056, 2160),
  'exposure_limits': (60, 127156999),
  'format': SRGGB12_CSI2P,
  'fps': 50.03,
  'size': (2028, 1080),
  'unpacked': 'SRGGB12'},
 {'bit_depth': 12,
  'crop_limits': (0, 0, 4056, 3040),
  'exposure_limits': (60, 127156999),
  'format': SRGGB12_CSI2P,
  'fps': 40.01,
  'size': (2028, 1520),
  'unpacked': 'SRGGB12'},
 {'bit_depth': 12,
  'crop_limits': (0, 0, 4056, 3040),
  'exposure_limits': (114, 239542228),
  'format': SRGGB12_CSI2P,
  'fps': 10.0,
  'size': (4056, 3040),
  'unpacked': 'SRGGB12'}]
```

This gives us the exact sensor modes that we can request, with the following information for each mode:

- `bit_depth` - the number of bits in each pixel sample.
- `crop_limits` - this tells us the exact field of view of this mode within the full resolution sensor output. In the example above, only the final two modes will give us the full field of view.
- `exposure_limits` - the maximum and minimum exposure values (in microseconds) permitted in this mode.
- `format` - the packed sensor format. This can be passed as the "format" when requesting the raw stream.
- `fps` - the maximum framerate supported by this mode.
- `size` - the resolution of the sensor output. This value can be passed as the "size" when requesting the raw stream.
- `unpacked` - use this in place of the earlier `format` in the raw stream request if unpacked raw images are required (see below).

In this example there are three 12-bit modes (one at the full resolution) and one 10-bit mode useful for higher framerate applications (but with a heavily cropped field of view). If you want to choose the raw mode yourself, only the "format" and "size" fields have any effect but the other fields will be ignored, meaning you can use statements like:

```
config = picam2.create_preview_configuration({"size": (640, 480)}, raw=picam2.sensor_modes[2])
```

NOTE

For a raw stream, the format normally begins with an `S`, followed by 4 characters that indicate the *Bayer order* of the sensor (the only exception to this is for raw *monochrome* sensors, which use the single letter `R` instead). Next is a number, 10 or 12 here, which is the number of bits in each pixel sample sent by the sensor (some sensors may have 8 bit modes too). Finally there *may* be the characters `_CSI2P`. This would mean that the pixel data will be packed tightly in memory, so that four 10-bit pixel values will be stored in every 5 bytes, or two 12-bit pixel values in every 3 bytes. When `_CSI2P` is absent, it means the pixels will be unpacked each one into a 16-bit word (or 8-bit pixels into a single byte). This uses more memory but can be useful for applications that want easy access to the raw pixel data.

4.2.3. Configurations and Runtime Camera Controls

The final element in a *Picamera2* configuration are runtime camera controls. As the description suggests, we normally appy these at runtime and they are not really part of the camera configuration. However, there are some occasions when it can be helpful to associate them directly with a "configuration".

One such example is in video recording. Normally the camera can run at a variety of framerates, and this can be changed by an application while the camera is running. When recording a video, however, people commonly prefer to record at a fixed 30 frames per second, even if the camera was set to something else previously.

The configuration-generating methods therefore supply some recommended runtime control values corresponding to the use case. These can be overridden or changed, but as the the optimal or usual values are sometimes a bit technical, it's helpful to supply them automatically:

```
>>> from picamera2 import Picamera2
>>> picam2 = Picamera2()
>>> picam2.create_video_configuration()["controls"]
{'NoiseReductionMode': <NoiseReductionMode.Fast: 1>, 'FrameDurationLimits': (33333, 33333)}
```

We see here that for a video use-case, we're recommended to set the *NoiseReductionMode* to `Fast` (because when encoding video it's important to get the frames quickly), and the *FrameDurationLimits* are set to `(33333, 33333)`. This means that every camera frame may not take less than the first value (33333 microseconds), and may not take longer than the second (also 33333 microseconds). Therefore the framerate is fixed to 30 frames per second.

New control values, or ones to override the default ones, can be specified with the `controls` keyword parameter. If you wanted 25 frames per second video you could use:

```
from picamera2 import Picamera2

picam2 = Picamera2()
config = picam2.create_video_configuration(controls={"FrameDurationLimits": (40000, 40000)})
```

When controls have been set into the returned configuration object, we can think of them as being part of that configuration. If we hold on to the configuration object and apply it again later, then those control values will be restored.

We are of course always free to set runtime controls later using the `Picamera2.set_controls` method, but these will not become part of any configuration that we can recover later.

For a full list of all the available runtime camera controls, please refer to the relevant appendix.

4.3. Configuration Objects

We have seen numerous examples of creating camera configurations using the `create_xxx_configuration` methods which return regular Python dictionaries. Some users may prefer the "object" style of syntax so we provide this as an alternative. Neither style is particularly recommended over the other.

Camera configurations can be represented by the `CameraConfiguration` class. This class contains the exact same things we have seen previously, namely:

- the `buffer_count`,
- a `Transform` object,
- a `ColorSpace` object,
- the name of the stream to display (if any),
- the name of the stream to encode (if any),
- a `controls` object which represents camera controls through a `Controls` class,
- a `main` stream,
- and optionally a `lores` and/or a `raw` stream.

When a `Picamera2` object is created, it contains three embedded configurations, in the following fields:

- `preview_configuration` - the same configuration as is returned by the `create_preview_configuration` method
- `still_configuration` - the same configuration as is returned by the `create_still_configuration` method

- video_configuration - the same configuration as is returned by the create_video_configuration method.

Finally, CameraConfiguration objects can also be passed to the configure method. Alternatively, the strings "preview", "still" and "video" may be passed as shorthand for the three embedded configurations listed above.

We conclude with some examples.

Changing the size of the main stream

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.preview_configuration.main.size = (800, 600)
picam2.configure("preview")
```

Here, the configuration is equivalent to that generated by picam2.create_preview_configuration({"size": (800, 600)}). As another shorthand, the main field can be elided, so picam2.preview_configuration.size = (800, 600) would have been identical.

Configuring a *lores* or *raw* stream

Before setting the size or format of the optional streams, they must first be enabled with:

```
configuration_object.enable_lores()
```

or

```
configuration_object.enable_raw()
```

as appropriate. This would normally be advised after the main stream size has been set up so that they can pick up more appropriate defaults. After that, the size and the format fields can be set in the usual way:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.preview_configuration.enable_lores()
picam2.preview_configuration.lores.size = (320, 240)
picam2.configure("preview")
```

Setting the format field is optional as defaults will be chosen - "YUV420" for the *lores* stream. In the case of the *raw* stream format, this can be left at its default value (None) and the system will use the sensor's native format.

Stream alignment

Just as with the dictionary method, stream sizes are not forced to optimal alignment by default. This can easily be accomplished using the configuration object's align method:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.preview_configuration.main.size = (808, 600)
picam2.preview_configuration.main.format = "YUV420"
picam2.preview_configuration.align()
picam2.configure("preview")
```

Supplying control values

We saw earlier how control values can be associated with a configuration. Using the object tyle of configuration, the equivalent to that example would be:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.video_configuration.controls.FrameDurationLimits = (40000, 40000)
picam2.configure("video")
```

For convenience, the "controls" object lets you set the `FrameRate` instead of the `FrameDurationLimits` in case this is easier. You can give it either a range (as `FrameDurationLimits`) or a single value, where *framerate* = 1000000 / *frameduration*, and *frameduration* is given in microseconds (as we did above):

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.video_configuration.controls.FrameRate = 25.0
picam2.configure("video")
```

We discuss setting control values at runtime in the following section.

4.4. Further Examples

Most of the GitHub examples will configure the camera, for example:

- capture_dng_and_jpeg.py - shows how you can configure a still capture for a full resolution *main* stream and also obtain the *raw* image buffer.
- capture_motion.py - shows how you can capture both a *main* and a *lores* stream.
- rotation.py - shows a 180 degree rotation being applied to the camera images. In this example the rotation is applied after the configuration has been generated, though we could have pass the transform in to the `create_preview_configuration` function with `transform=libcamera.Transform(hflip=1, vflip=1)` too.
- still_capture_with_config.py - shows how to configure a still capture using the configuration object method. In this example we also request a *raw* stream.

Chapter 5. Camera Controls and Properties

5.1. Camera Controls

Camera controls represent parameters that the camera exposes, and which we can alter to change the nature of the images it outputs in various ways.

In *Picamera2*, all camera controls can be changed at runtime. Anything that cannot be changed at runtime is regarded not as a control but as *configuration*. We do, however, allow the camera's configuration to include controls in case there are particular standard control values that could be conveniently applied along with the rest of the configuration.

For example, some obvious controls that we might want to set while the camera is delivering images are:

- Exposure time
- Gain
- White balance
- Colour saturation
- Sharpness

and there are many more. A complete list of all the available camera controls can be found in appendix *TODO: add link* and also by inspecting the `camera_controls` property of the `Picamera2` object:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.camera_controls
```

This returns a dictionary with the control names as keys, and each value being a *tuple* of *(min, max, default)* values for that control. The default value should be interpreted with some caution as in many cases *libcamera*'s default value will long since have been overwritten by the camera tuning once the camera is started.

Things that are *not* controls include:

- The image resolution
- The format of the pixels

as these can only be set up when the camera is configured.

One example of a control that might be assoicated with a configuration might be the camera's *framerate* (or equivalently, the *frame duration*). Normally we might let a camera operate at whatever framerate is appropriate to the exposure time requested. For video recording, however, it's quite common to fix the framerate to (for example) 30 frames per second, and so this might be included by default along with the rest of the video configuration.

5.1.1. How to set Camera Controls

There a three distinct times when camera controls can be set:

1. Into the camera configuration. These will be stored with the configuration so that they will be re-applied whenever that configuration is requested. They will be enacted before the camera starts.

2. After configuration but before the camera starts. The controls will again take effect before the camera starts, but will not be stored with the configuration and so would not be re-applied again automatically.
3. After the camera has started. The camera system will apply the controls as soon as it can, but typically there will be some number of frames of delay.

Camera controls can be set by passing a dictionary of updates to the `set_controls` method, but there is also an object style syntax for accomplishing the same thing.

5.1.1.1. Setting Controls as part of the Configuration

We have seen an example of this when discussing camera configurations. One important feature of this is that the controls are applied before the camera even starts, meaning the the *very first camera frame* will have the controls set as requested.

5.1.1.2. Setting Controls before the Camera starts

This time we set the controls after configuring the camera but before starting it. For example:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration())
picam2.set_controls({"ExposureTime": 10000, "AnalogueGain": 1.0})
picam2.start()
```

Here too the controls will have already been applied on the very first frame that we receive from the camera.

5.1.1.3. Setting Controls after the Camera has started

This time, there will be a delay of several frames before the controls take effect. This is because there is perhaps quite a large number of requests for camera frames already in flight, and for some controls (exposure time and analogue gain specifically), the camera may actually take several frames to apply the updates.

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration())
picam2.start()
picam2.set_controls({"ExposureTime": 10000, "AnalogueGain": 1.0})
```

This time we cannot rely on any specific frame having the value we want and would have to check the frame's metadata.

5.1.2. Object Syntax for Camera Controls

We saw previously how control values can be associated with a particular camera configuration.

There is also an embedded instance of the `Controls` class inside the `Picamera2` object that allows controls to be set subsequently. For example, to set controls after configuration but before starting the camera:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.configure("preview")
picam2.controls.ExposureTime = 10000
picam2.controls.AnalogueGain = 1.0
picam2.start()
```

To set these controls after the camera has started we should use:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.configure("preview")
picam2.start()
with picam2.controls as controls:
    controls.ExposureTime = 10000
    controls.AnalogueGain = 1.0
```

In this final case we note the use of the `with` construct. Although you would normally get by without it (just set the `picam2.controls` directly), that would not absolutely guarantee that both controls would be applied on the same frame. You could technically find the analogue gain being set on the frame after the exposure time.

In all cases, the same rules apply as to whether the controls take effect immediately or incur several frames of delay.

5.2. Camera Properties

Camera properties represent information about the sensor that applications may want to know. They cannot be changed by the application at any time, neither at runtime nor while configuring the camera, although the value of these properties *may* change whenever the camera is configured.

Camera properties may be inspected through the `camera_properties` property of the `Picamera2` object:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.camera_properties
```

Some examples of camera properties include the model of sensor and the size of the pixel array. After configuring the camera into a particular mode it will also report the field of view from the pixel array that the mode represents, and the sensitivity of this mode relative to other camera modes.

A complete list and explanation of each property can be found in appendix *TODO: add link*.

5.3. Further Examples

The following examples demonstrate setting controls:

- controls.py - shows how to set controls while the camera is running. In this example we query the `ExposureTime`, `AnalogueGain` and `ColourGains` and then fix them so that they can no longer vary.
- opencv_mertens_merge.py - demonstrates how to stop and restart the camera with multiple new exposure values.

- zoom.py - shows how to implement a smooth digital zoom. We use `capture_metadata` to synchronise the control value updates with frames coming from the camera.
- controls_3.py - illustrates the use of the `Controls` class, rather than dictionaries, to update control values.

Chapter 6. Capturing Images and Requests

Once the camera has been configured and started, *Picamera2* automatically starts submitting *requests* to the camera system. Each request will be completed and returned to *Picamera2* once the camera system has filled in an image buffer for each of the streams that were configured.

The process of requests being submitted, returned and then sent back to the camera system is transparent to the user. In fact, the process of sending the images for display (when a preview window has been set up) or forwarding them to a video encoder (when one is running) is all entirely automatic too, and the application does not have to do anything.

The user application only needs to say when it wants to receive any of these images for its own use and *Picamera2* will deliver them. The application can request a single image belonging to any of the streams, or it can ask for the entire request, giving access to all the images and the associated metadata.

NOTE

In this section we make use of some convenient default behaviour of the `start` function. If the camera is completely unconfigured, it will apply the usual default preview configuration before starting the camera.

6.1. Capturing Images

Camera images are normally represented as *numpy* arrays, so some familiarity with *numpy* will be helpful. This is also the representation used by *OpenCV* so that *Picamera2*, *numpy* and *OpenCV* all work together seemlessly.

When capturing images, the *Picamera2* functions use the following nomenclature in its capture functions:

- *arrays* - these are 2-dimensional arrays of pixels and are usually the most convenient way to manipulate images. They are often 3-dimensional *numpy* arrays because every pixel has several colour components, adding another dimension.
- *images* - this refers to *PIL* (*Python Image Library*) images and can be useful when interfacing to other modules that expect this format, and
- *buffers* - by *buffer* we simply mean the entire block of memory where the image is stored as a 1-dimensional *numpy* array, but the 2 (or 3) dimensional *array* form is generally more useful.

There are also capture functions for saving images directly to files, and for switching between camera modes so as to combine fast framerate previews with high resolution captures.

6.1.1. Capturing Arrays

The `capture_array` function will capture the next camera image from the stream named as its first argument (and which defaults to `"main"` if omitted). The following example starts the camera with a typical preview configuration, waits for one second (during which the camera is running), and then capture a 3-dimensional *numpy* array:

```
from picamera2 import Picamera2
import time

picam2 = Picamera2()
picam2.start()

time.sleep(1)
```

```
array = picam2.capture_array("main")
```

Although we regard this as a 2-dimensional image *numpy* will often report a 3rd dimension of size 3 or 4 depending on whether every pixel is represented by 3 channels (RGB) or 4 channels (RGBA, that is RGB with an alpha channel). Remember also that *numpy* lists the height as the *first* dimension.

- `shape` will report *(height, width, 3)* for 3 channel RGB type formats,
- `shape` will report *(height, width, 4)* for 4 channel RGBA (alpha) type formats, and
- `shape` will report *(height * 3 / 2, width)* for YUV420 formats.

YUv420 is a slightly special case because the first *height* rows give the Y channel, the next *height / 4* rows contain the U channel and the final *height / 4* rows contain the V channel. For the other formats, where there is an "alpha" value it will take the fixed value 255.

6.1.2. Capturing *PIL* Images

PIL images are captured identically, but using the `capture_image` function.

```
from picamera2 import Picamera2
import time

picam2 = Picamera2()
picam2.start()

time.sleep(1)
image = picam2.capture_image("main")
```

6.1.3. Switching Camera Mode and capturing

A common use case is to run the camera in a mode where it can achieve a fast framerate for display in a preview window, but to switch to a (slower framerate) high resolution capture mode for the best quality images. There are special `switch_mode_and_capture_array` and `switch_mode_and_capture_image` functions for this.

```
from picamera2 import Picamera2
import time

picam2 = Picamera2()
capture_config = picam2.create_still_configuration()
picam2.start(show_preview=True)

time.sleep(1)
array = picam2.switch_mode_and_capture_array(capture_config, "main")
```

This will switch to the high resolution capture mode and return the *numpy* array, and will then switch automatically back to the preview mode without any user intervention.

We note that the process of switching camera modes can be performed "manually", if preferred:

```
from picamera2 import Picamera2
import time

picam2 = Picamera2()
```

```
preview_config = picam2.create_preview_configuration()
capture_config = picam2.create_still_configuration()
picam2 = picam2.configure(preview_config)
picam2.start(show_preview=True)

time.sleep(1)
picam2.switch_mode(capture_config)
array = picam2.capture_array("main")
picam2.switch_mode(preview_config)
```

6.1.4. Capturing straight to Files and file-like Objects

For convenience, the *capture* functions all have counterparts that save an image straight to file:

```
from picamera2 import Picamera2
import time

picam2 = Picamera2()
capture_config = picam2.create_still_configuration()
picam2.start(show_preview=True)

time.sleep(1)
picam2.switch_mode_and_capture_file(capture_config, "image.jpg")
```

The file format is deduced automatically from the filename. *Picamera2* uses *PIL* to save the images, and so this supports JPEG, BMP, PNG and GIF files.

But applications can also capture to file-like objects. A typical example would be memory buffers from Python's *io* library. In this case there is no filename so the format of the "file" must be given by the `format` parameter:

```
from picamera2 import Picamera2
import io
import time

picam2 = Picamera2()
picam2.start()

time.sleep(1)
data = io.BytesIO()
picam2.capture_file(data, format='jpeg')
```

The `format` parameter may take the values `'jpeg'`, `'png'`, `'bmp'` or `'gif'`.

6.2. Capturing Metadata

Every image that is output by the camera system comes with *metadata* that describes the conditions of the image capture. Specifically, the metadata describes the camera controls that were used to capture that image, including, for example, the exposure time and analogue gain of the image.

Metadata is returned as a Python dictionary and is easily captured by:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.start()

metadata = picam2.capture_metadata()
print(metadata["ExposureTime"], metadata["AnalogueGain"])
```

Capturing metadata is a good way to synchronise an application with camera frames (if you have no actual need of the frames). The first call to `capture_metadata` (or indeed any capture function) will often return immediately because *Picamera2* usually holds on to the last camera image internally. But after that, every capture call will wait for a new frame to arrive (unless the application has waited so long to make the request that the image is once again already there). For example:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.start()

for i in range(30):
    metadata = picam2.capture_metadata()
    print("Frame", i, "has arrived")
```

The process of obtaining the metadata that belongs to a specific image is explained through the use of requests, just below.

Object-style Access to Metadata

For those who prefer the object-style syntax over Python dictionaries, the metadata can be wrapped in the `Metadata` class:

```
from picamera2 import Picamera2, Metadata

picam2 = Picamera2()
picam2.start()

metadata = Metadata(picam2.capture_metadata())
print(metadata.ExposureTime, metadata.AnalogueGain)
```

6.3. Capturing Multiple Images at Once

A good way to do this is by capturing an entire request, but sometimes it is conveient to be able to obtain copies of multiple *numpy* arrays in much the same way as we were able to capture a single one.

For this reason some of the functions we saw earlier have "pluralised" versions. To list them explicitly:

Capture single array	Capture multiple arrays
`Picamera2.capture_buffer`	`Picamera2.capture_buffers`
`Picamera2.switch_mode_and_capture_buffer`	`Picamera2.switch_mode_and_capture_buffers`
`Picamera2.capture_array`	`Picamera2.capture_arrays`
`Picamera2.switch_mode_and_capture_array`	`Picamera2.switch_mode_and_capture_arrays`

All these functions work in the same way as their single-capture counterparts except that:

- Instead of providing a single stream name, a list of stream names must be provided.
- The return value is a *tuple* of two values, the first being the list of arrays (in the order the names were given), and the second being the image metadata.

For example we could use:

```
from picamera2 import Picamera2

picam2 = Picamera2()
config = picam2.create_preview_configuration(lores={})
picam2.configure(config)
picam2.start()

(main, lores), metadata = picam2.capture_arrays(["main", "lores"])
```

In this case we configure both a *main* and a *lores* stream. We then ask to capture an image from each and these are returned to us along with the metadata for that single capture from the camera.

Finally, to faciliate using these images, *Picamera2* has a small *Helpers* library that can convert arrays to PIL images, save them to a file, and so on. The table below lists all the available functions:

Helper name	Description
`picam2.helpers.make_array`	Make a 2d (or 3d, allowing for multiple colour channels) array from a flat 1d array (as returned by, for example, `capture_buffer`).
`picam2.helpers.make_image`	Make a PIL image from a flat 1d array (as returned by, for example, `capture_buffer`).
`picam2.helpers.save`	Save a PIL image to file.
`picam2.helpers.save_dng`	Save PIL image to a DNG file.

These helpers can be accessed directly from the *Picamera2* object. If we wanted to capture a single buffer and use one of these helpers to save the file, we could use:

```
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration())
picam2.start()

(buffer, ), metadata = picam2.capture_buffers(["main"])
img = picam2.helpers.make_image(buffer, picam2.camera_configuration()["main"])
picam2.helpers.save(img, metadata, "file.jpg")
```

Further examples are highlighted at the end of this chapter.

6.4. Capturing Requests

Besides capturing images from just one of the configured streams, or the image metadata, *Picamera2* makes it possible to capture an *entire request*. This includes the image from every stream that has been configured, and also the metadata, so that they can be used together.

Normally, when we capture arrays or images, the image data is copied so that the camera system can keep hold of all the memory buffers it was using and carry on running in just the same manner. When we capture a request, however, we have only *borrowed* it, and all the memory buffers, from the camera system, and nothing has yet been copied. When we are finished with the request it must be returned back to the camera system using the request's `release` method.

IMPORTANT

If an application fails to release captured requests back to the camera system, the camera system will gradually run out of buffers. It is likely to start dropping ever more camera frames, and eventually the camera system will stall completely.

Here is the basic use pattern:

```
from picamera2 import Picamera

picam2 = Picamera2()
picam2.start()

request = picam2.capture_request()
request.save("main", "image.jpg")
print(request.get_metadata())  # this is the metadata for this image
request.release()
```

As soon as we have finished with the request, it is released back to the camera system. This example also shows how we are able to obtain the exact metadata for the captured image using the request's `get_metadata` function.

Notice how we saved the image from the *main* stream to the file. All *Picamera2*'s capture methods are implemented through methods in the `CompletedRequest` class, and once we have the request we can call them directly. The correspondance is illustrated in the table below.

Picamera2 **function**	*CompletedRequest* **equivalent**
`Picamera2.capture_file`	`CompletedRequest.save` (or `CompletedRequest.save_dng` for raw files)
`Picamera2.capture_buffer`	`CompletedRequest.make_buffer`
`Picamera2.capture_array`	`CompletedRequest.make_array`
`Picamera2.capture_image`	`CompletedRequest.make_image`

Moving processing out of the Camera Thread

Normally when we use a function like `Picamera2.capture_file`, the processing to capture the image, compress it as (for example) a JPEG, and save it to file happens in the usual camera processing loop. While this happens, the handling of camera events is blocked and the camera system is likely to drop some frames. In many cases this is not terribly important, but there are occasions when we might prefer all the processing to happen somewhere else.

Just as an example, if we were recording a video and wanted to capture a JPEG simultaneously whilst minimising the risk of dropping any video frames, then it would be beneficial to move that processing out of the camera loop.

This is easily accomplished simply by capturing a request and calling `request.save` as we saw above. Camera events can still be handled in parallel (though this is somewhat at the mercy of Python's multi-tasking abilities), and the only downside is that camera system has to make do with one fewer set of buffers until that request is finally released. However, this can in turn always be mitigated by allocating one or more extra sets of buffers via the camera configuration's `buffer_count` parameter.

6.5. Asynchronous Capture

Sometimes it's convenient to be able to ask *Picamera2* to capture an image (for example), but not to block your thread while this takes place. Among the advanced use cases later on we'll see some particular examples of this, although it's a feature that might be helpful in other circumstances too.

All the *capture* and *switch_mode_and_capture* methods take two additional arguments:

- `wait` - whether to block while the operation completes, and
- `signal_function` - a function that will be called when the operation completes.

Both take the default value `None`, but can be changed resulting in the following behaviour:

- If `wait` and `signal_function` are both `None`, then the function will block until the operation is complete. This is what we would probably term the "usual" behaviour.
- If `wait` is `None` but a `signal_function` is supplied, then the function will *not* block, but return immediately even though the operation is not complete. The caller should use the supplied `signal_function` to notify the application when the operation is complete.
- If a `signal_function` is supplied, and `wait` is not `None`, then the given value of `wait` determines whether the function blocks (it blocks if `wait` is true). The `signal_function` is still called, however.
- You can also set `wait` to `False` and not supply a `signal_function`. In this case the function returns immediately, and you can block later for the operation to complete (see below).

WARNING

The `signal_function` should *not* initiate any *Picamera2* activity (by calling *Picamera2* methods, for example) itself, as this is likely to result in a deadlock. Instead, it should be setting events or variables for threads to respond to.

Completing an Asynchronous Request

After launching an asynchronous operation as described above, an application must always call `Picamera2.wait()` to complete the process, for example:

```
result = picam2.wait()
```

The `wait` function returns the result that would have been returned if the operation had blocked initially. It is mandatory to call `wait` before another *Picamera2* capture request can be made, so there can only be one asynchronous request in flight at once.

Here's a short example. The `switch_mode_and_capture_file` method captures an image to file and returns the image metadata. So we can do the following:

```
from picamera2 import Picamera2
import time

picam2 = Picamera2()
still_config = picam2.create_still_configuration()
picam2.configure(picam2.create_preview_configuration())
picam2.start()
time.sleep(1)
picam2.switch_mode_and_capture_file(still_config, "test.jpg", wait=False)

# now we can do some other stuff...
for i in range(20):
    time.sleep(0.1)
    print(i)

# finally complete the operation:
metadata = picam2.wait()
```

6.6. High Level Capture API

There are some high level image capture functions provided for those who may not want such an in-depth understanding of how *Picamera2* works. These functions should not be called from the camera's event processing thread (for example, it could be called directly from the Python interpreter, possibly via a script).

6.6.1. start_and_capture_file

For simple image capture we have the `Picamera2.start_and_capture_file` method. This function will configure and start the camera automatically, and return once the capture is complete. It accepts the following parameters though all have sensible default values so that the function can be called with no arguments at all.

- `name` (default `"image.jpg"`) - the file name under which to save the captured image.
- `delay` (default `1`) - the number of seconds of delay before capturing the image. The value zero (no delay) is valid.
- `preview_mode` (default `"preview"`) - the camera configuration to use for the preview phase of the operation. The default value indicates to use the configuration in the *Picamera2* object's `preview_configuration` field, though any other configuration can be supplied. The capture operation only has a preview phase if the `delay` is greater than zero.
- `capture_mode` (default `"still"`) - the camera configuration to use for capturing the image. The default value indicates to use the configuration in the *Picamera2* object's `still_configuration` field, though any other configuration can be supplied.
- `show_preview` (default `True`) - whether to show a preview window. By default preview images are only displayed for the preview phase of the operation, unless this behaviour is overridden by the supplied camera configurations using the `display` parameter. If subsequent calls are made which *change* the value of this parameter, we note that the application should call the `Picamera2.stop_preview` method in between.

This function can be used as follows:

```python
from picamera2 import Picamera2

picam2 = Picamera2()

picam2.start_and_capture_file("test.jpg")
```

All the usual file formats, JPEG, PNG, BMP and GIF, are supported.

6.6.2. start_and_capture_files

This function is very similar to `start_and_capture_file`, except that it can capture multiple images with a time delay between them. Again, it can be called with no arguments at all, but it accepts the following optional parameters:

- `name` (default `"image{:03d}.jpg"`) - the file name under which to save the captured image. This should include a format directive (such as in the default name) that will be replaced by a counter, otherwise the images will simply over-write one another.
- `initial_delay` (default `1`) - the number of seconds of delay before capturing the first image. The value zero (no delay) is valid.
- `preview_mode` (default `"preview"`) - the camera configuration to use for the preview phases of the operation. The default value indicates to use the configuration in the *Picamera2* object's `preview_configuration` field, though any other configuration can be supplied. The capture operation only has a preview phase when the corresponding delay parameter (`delay` or `initial_delay`) is greater than zero.

- capture_mode (default "still") - the camera configuration to use for capturing the images. The default value indicates to use the configuration in the *Picamera2* object's still_configuration field, though any other configuration can be supplied.
- num_files (default 1) - the number of images to capture.
- delay (default 1) - the number of second between captures for all images except the very first (which is governed by initial_delay). The this has the value zero, then there is no preview phase between the captures at all.
- show_preview (default True) - whether to show a preview window. By default preview images are only displayed for the preview phases of the operation, unless this behaviour is overridden by the supplied camera configurations using the display parameter. If subsequent calls are made which *change* the value of this parameter, we note that the application should call the Picamera2.stop_preview method in between.

Finally, it could be used as follows:

```
from picamera2 import Picamera2

picam2 = Picamera2()

picam2.start_and_capture_files("test{:d}.jpg", initial_delay=5, delay=5, num_files=10)
```

This will capture 10 files named test0.jpg through test9.jpg, with a 5 second delay before every capture.

To capture images back-to-back with minimum delay, one could use:

```
from picamera2 import Picamera2

picam2 = Picamera2()

picam2.start_and_capture_files("test{:d}.jpg", initial_delay=0, delay=0, num_files=10)
```

In practice the rate of capture will be limited by the time it takes to encode and save the JPEG files. For faster capture, it might be worth considering saving a video in MJPEG format instead.

6.7. Further Examples

Many of the examples demonstrate how to capture images. We draw the reader's attention to just a few of them:

- metadata_with_image.py - shows how to capture an image and also the metadata for that image.
- capture_to_buffer.py - shows how to capture to a buffer rather than a file.
- still_during_video.py - shows how you might capture a still image while a video recording is in progress.
- opencv_mertens_merge.py - takes several captures at different exposures, starting and stopping the camera for each capture.
- easy_capture.py - uses the "high level" API to capture images.
- capture_dng_and_jpeg_helpers.py - uses the "helpers" to save a JPEG and DNG file without holding the entire CompletedRequest object.

Chapter 7. Capturing Videos

In *Picamera2*, the process of capturing and encoding video is largely automatic. The application only has to define what *encoder* it wants to use to compress the image data, and how it wants to *output* this compressed data stream.

The mechanics of taking the camera images that arrive, forwarding them to an encoder, which in turn sends the results directly to the requested output, is entirely transparent to the user. The encoding and output all happens in a separate thread from the camera handling so as to minimise the risk of dropping camera frames.

Here is a first example of capturing a 10 second video.

```
from picamera2.encoders import H264Encoder
from picamera2 import Picamera2
import time

picam2 = Picamera2()
video_config = picam2.create_video_configuration()
picam2.configure(video_config)

encoder = H264Encoder(bitrate=10000000)
output = "test.h264"

picam2.start_recording(encoder, output)
time.sleep(10)
picam2.stop_recording()
```

In this example we use the H.264 encoder. For the output object we can just use a string for convenience; this will be intpreted as a simple output file. For configuring the camera, the `create_video_configuration` is a good starting point as it will use a larger `buffer_count` to reduce the risk of dropping frames.

We also used the convenient `start_recording` and `stop_recording` functions, which start and stop both the encoder and the camera together. Sometimes it can be useful to separate these two operations, for example you might want to start and stop a recording multiple times while leaving the camera running throughout. For this reason, `start_recording` could have been replaced by:

```
encoder.output = 'test.h264'
picam2.start_encoder(encoder)
picam2.start()
```

and `stop_recording` by:

```
picam2.stop()
picam2.stop_encoder()
```

7.1. Encoders

All the video encoders can be constructed with parameters that determine the quality (amount of compression) of the output, such as the `bitrate` for the H.264 encoder. For those not so familiar with the details of these encoders, these parameters can also be omitted in favour of supplying a *quality* to the `start_encoder` or `start_recording` functions. The permitted quality parameters are:

- `Quality.VERY_LOW`
- `Quality.LOW`
- `Quality.MEDIUM` - this is the default for both functions if the parameter is not specified.
- `Quality.HIGH`
- `Quality.VERY_HIGH`

This quality parameter only has any effect if the encoder was not passed explicit codec-specific parameters. It could be used like this:

```
from picamera2.encoders import H264Encoder, Quality
from picamera2 import Picamera2
import time

picam2 = Picamera2()
picam2.configure(picam2.create_video_configuration())

encoder = H264Encoder()

picam2.start_recording(encoder, 'test.h264', Quality.HIGH)
time.sleep(10)
picam2.stop_recording()
```

Suitable adjustments will be made by the encoder according to the supplied quality parameter, though this is of a "best efforts" nature and somewhat subject to interpretation. Applications are recommended to choose explicit parameters for themselves if the quality parameter is not having the desired effect.

The available encoders are described in the sections that follow.

7.1.1. H264Encoder

The `H264Encoder` class implements an H.264 encoder using the Pi's in-built hardware, accessed through the V4L2 kernel drivers, supporting up to 1080p30. The constructor accepts the following optional parameters:

- `bitrate` (default `None`) - the bitrate (in bits per second) to use. The default value `None` will cause the encoder to choose an appropriate bitrate according to the `Quality` when it starts.
- `repeat` (default `False`) - whether to repeat the stream's sequence headers with every I frame. This can be sometimes be useful when streaming video over a network, when the client may not receive the start of the stream where the sequence headers would normally be located.
- `iperiod` (default `None`) - the number of frames from one I (intra) frame to the next. The value `None` leaves this at the discretion of the hardware, which defaults to 60 frames.

This encoder can accept either 3-channel RGB (`"RGB888"` or `"BGR888"`), 4-channel RGBA (`"XBGR8888"` or `"XRGB8888"`) or YUV420 (`"YUV420"`). *TODO: it probably accepts more YUV formats, not sure which*

7.1.2. JpegEncoder

The `JpegEncoder` class implements a multi-threaded software JPEG encoder, which can also be used as a motion JPEG ("MJPEG") encoder. It accepts the following optional parameters:

- `num_threads` (default `4`) - the number of concurrent threads to use for encoding.
- `q` (default `None`) - the JPEG quality number. The default value `None` will cause the encoder to choose an appropriate value according to the `Quality` when it starts.

- `colour_space` (default `RGBX`) - the colour space parameter that needs to be passed to the JPEG encoder library to match the pixel format of the stream being encoded. *TODO: this actually needs fixing in the software so that it's chosen automatically*

This encoder can accept either 3-channel RGB (`"RGB888"` or `"BGR888"`) or 4-channel RGBA (`"XBGR8888"` or `"XRGB8888"`).

NOTE

The `JpegEncoder` is derived from the `MultiEncoder` class which wraps frame-level multi-threading around an existing software encoder, and some users may find it helpful in creating their own parallelised codec implementations. There will only be a significant performance boost if Python's *GIL* (*Global Interpreter Lock*) can be released while the encoding is happening - as is the case with the `JpegEncoder` as the bulk of the work happens in a call to a C library.

7.1.3. MJPEGEncoder

The `MJPEGEncoder` class implements an MJPEG encoder using the Pi's in-built hardware, accessed through the V4L2 kernel drivers. The constructor accepts the following optional parameter:

- `bitrate` (default `None`) - the bitrate (in bits per second) to use. The default value `None` will cause the encoder to choose an appropriate bitrate according to the `Quality` when it starts.

This encoder can accept either 3-channel RGB (`"RGB888"` or `"BGR888"`), 4-channel RGBA (`"XBGR8888"` or `"XRGB8888"`) or YUV420 (`"YUV420"`). *TODO: it probably accepts more YUV formats, not sure which*

7.2. Outputs

Output objects receive encoded video frames directly from the encoder and will typically forward them to files or to network sockets. An output object is often made with its constructor, although a simple string can be passed to the `start_encoder` and `start_recording` functions which will cause a `FileOutput` object to be made automatically.

The available output objects are described in the sections that follow.

7.2.1. FileOutput

The `FileOutput` is constructed from a single `file` parameter which may be:

- the value `None`, which causes any output to be discarded,
- a string, in which case a regular file is opened, or
- a file-like object, which might be a memory buffer created using `io.BytesIO()`, or a network socket.

For example, a simple file output could be created with:

```
from picamera2.outputs import FileOutput

output = FileOutput('test.h264')
```

A memory buffer:

```
from picamera2.outputs import FileOutput
import io

buffer = io.Bytes()
output = FileOutput(buffer)
```

A UDP network socket:

```
from picamera2.outputs import FileOutput
import socket

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
    sock.connect(("REMOTEIP", 10001))
    stream = sock.makefile("wb")
    output = FileOutput(stream)
```

7.2.2. FfmpegOutput

The `FfmpegOutput` class forwards the encoded frames to an *FFmpeg* process. This opens the door to some quite sophisticated new kinds of output, including *mp4* files and even audio, but may require increasing amounts of knowledge about *FFmpeg* itself (which is well beyond the scope of this document, but *FFmpeg*'s own documentation is available).

The class constructor has one required parameter, the output file name, and all the others are optional:

- `output_filename` - typically we might pass something like `"test.mp4"`, however, it is used as the output part of the *FFmpeg* command line and so could equally well contain `"test.ts"` (to record an MPEG-2 Transport Stream), or even `"-f mpegts udp://<ip-addr>:>port"` to forward an MPEG-2 Transport Stream to a given network socket.
- `audio` (default `False`) - if you have an attached microphone, pass `True` for the audio to be recorded along with the video feed from the camera. The microphone is assumed to be available through Pulseaudio.
- `audio_device` (default `"default"`) - the name by which Pulseaudio knows the microphone. Usually `"default"` will work.
- `audio_sync` (default `-0.3`) - the time shift in seconds to apply between the audio and video streams. This may need tweaking to improve the audio/video synchronisation.
- `audio_samplerate` (default `48000`) - the audio sample rate to use.
- `audio_codec` (default `"aac"`) - the audio codec to use.
- `audio_bitrate` (default `128000`) - the bitrate for the audio codec.

The range of output file names that can be passed to *FFmpeg* is very wide because it may actually include any of *FFmpeg*'s output options, thereby exceeding the scope of what could be documentated here or even tested comprehensively. We list some more complex examples later on, and conclude with a simple example that records audio and video to an *mp4* file:

```
from picamera2.outputs import FfmpegOutput

output = FfmpegOutput("test.mp4", audio=True)
```

NOTE

One limitation of the `FfmpegOutput` class is that we don't know of an easy way to pass the frame timestamps - which *Picamera2* knows precisely - to *FFmpeg*. As such, we have to get *FFmpeg* to resample them, meaning they become subject to a relatively high degree of jitter. Whilst this may matter for some applications, it does not affect most users.

7.2.3. CircularOutput

The `CircularOutput` class is derived from the `FileOutput` and adds the ability to start a recording with video frames that were from several seconds earlier. This is ideal for motion detection and security applications. The `CircularOutput` constructor accepts the following optional parameters:

- `file` (default `None`) - a string (representing a file name) or a file-like object which is used to construct a `FileOutput`. This is where output from the circular buffer will get written. The value `None` means that, when the circular buffer is created, it will accumulate frames within the circular buffer, but will not be writing them out anywhere.
- `buffersize` (default `150`) - set this to the number of seconds of video you want to be able to access from before the current time, multiplied by the framerate. So 150 buffers is enough for 5s of video at 30fps.

To make the `CircularOutput` start writing the frames out to a file (for example), an application should:

1. set the `fileoutput` property of the `CircularOutput` object, and
2. call its `start()` method.

In outline, the code will look something like this:

```python
from picamera2.encoders import H264Encoder
from picamera2.outputs import CircularOutput
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.configure(picam2.create_video_configuration())

encoder = H264Encoder()
output = CircularOutput()

picam2.start_recording(encoder, output)

# Now when it's time to start recording the output, including the previous 5 seconds:
output.fileoutput = "file.h264"
output.start()

# And later it can be stopped with:
output.stop()
```

7.3. High Level Video Recording API

As we saw with still captures, video recording also has a convenient API for those who wish to know less about the encoders and output objects that make it work. We have the function `start_and_record_video`, which takes a file name as a required argument and a further number of optional ones:

- `output` (required) - the name of the file to record to, or an output object of one of the types described above. When a string ending in `.mp4` is supplied, an `FfmpegOutput` rather than a `FileOutput` is created, so that a valid *mp4* file is made.

- encoder (default None) - the encoder to use. If left unspecified, the function will make a best effort to choose (MJPEG if the file name ends in *mjpg* or *mjpeg*, otherwise H.264).
- config (default None) - the camera configuration to use if not None. If the camera is unconfigured but none was given, the camera will be configured according to the "video" (Picamera2.video_configuration) configuration.
- quality (default Quality.MEDIUM) - the video quality to generate, unless overriden in the encoder object.
- show_preview (default False) - whether to show a preview window. If this value is changed, it will have no effect unless stop_preview is called beforehand.
- duration (default 0) - the duration for which to record the vide. The function will block for this long before stopping the recording. When the value is zero, the function returns immediately and the application will have to call stop_recording later.
- audio (default False) - whether to record an audio stream. This only works when recording to an *mp4* file, and a microphone is attached as the default Pulseaudio input.

This example will record a 5 second *mp4* file:

```
from picamera2 import Picamera2

picam2 = Picamera2()

picam2.start_and_record_video("test.mp4", duration=5)
```

7.4. Further Examples

- audio_video_capture.py - captures both audio and video streams to an *mp4* file. Obviously a microphone is required for this to work.
- capture_circular.py - demonstrates how to capture to a circular buffer when motion is detected.
- capture_circular_stream.py - is a similar but more complex example that simultaneously sends the stream over a network connection, using the multiple outputs feature.
- capture_mjpeg.py - shows how to capture an MJPEG stream.
- mjpeg_server.py - implements a simple web server than can deliver streaming MJPEG video to a web page.

Chapter 8. Advanced Topics

8.1. Display Overlays

All the *Picamera2* preview windows support *overlays*. That is a bitmap with an alpha channel that can be super-imposed over the live camera image. The alpha channel allows the overlay image to be opaque, partially transparent or wholly transparent, pixel by pixel.

To add an overlay we can use the `Picamera2.add_overlay` function. It takes a single argument which is a 3-dimensional *numpy* array. The first two dimensions are the height and then the width, and the final dimension should have the value 4 as all pixels have R, G, B and A (alpha) values.

We note that:

- The overlay width and height do not have to match the camera images being displayed as the overlay will be resized to fit exactly over the camera image.
- `set_overlay` should only be called after the camera has been configured, as only at this point does *Picamera2* know how large the camera images being displayed will be.
- The overlay is always copied by the `set_overlay` call, so it is safe for an application to overwrite the overlay afterwards.
- Overlays are designed to provide simple effects or GUI elements over a camera image. They are not designed for sophisticated or fast moving animations.
- Overlays ignore any *display transform* that was specified when the preview was created.

Here is a very simple example:

```
from picamera2 import Picamera2
import numpy as np

picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration())
picam2.start(show_preview=True)

overlay = np.zeros((300, 400, 4), dtype=np.uint8)
overlay[:150, 200:] = (255, 0, 0, 64)  # reddish
overlay[150:, :200] = (0, 255, 0, 64)  # greenish
overlay[150:, 200:] = (0, 0, 255, 64)  # blueish
picam2.set_overlay(overlay)
```

and the result shows the red, green and blue quadrants over the camera image:

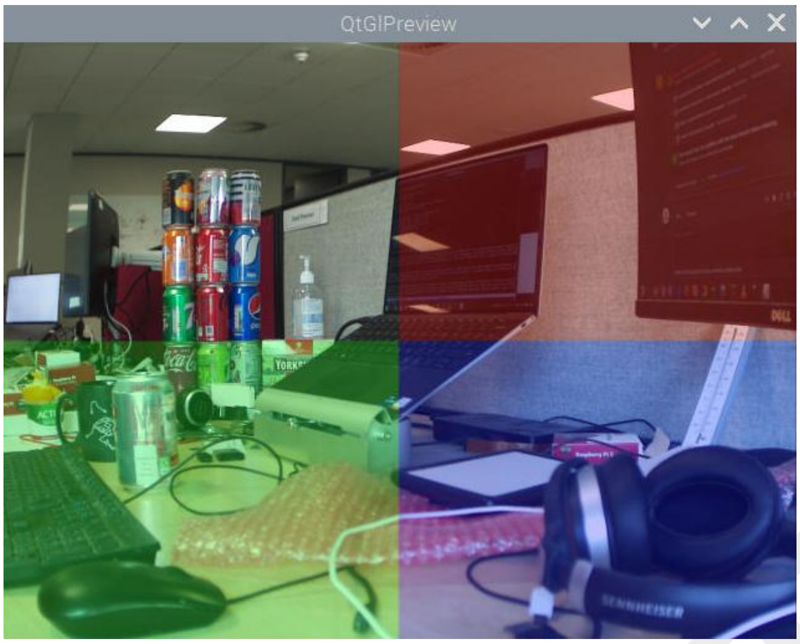

Figure 3. A simple overlay

For real applications, more complex overlays can of course be designed with image editing programs and loaded from file. Recall that, if loading an RGBA image with *OpenCV*, you need to use the `IMREAD_UNCHANGED` flag:

```
overlay = cv2.imread("overlay.png", cv2.IMREAD_UNCHANGED)
```

Further Examples

You can find the simple overlay example in:

- overlay_gl.py for the QTGL version of the overlay code.
- overlay_qt.py for the QT version of the overlay code.
- overlay_drm.py for the DRM version of the overlay code.

8.2. The Event Loop

8.2.1. Using the Event Loop Callbacks

We saw in the discussion of preview windows how the preview window normally supplies the *event loop* that feeds image buffers into *libcamera* in the form of requests, and receives tham back again once they are completed. Even when there is no actual preview window, we start the `NullPreview` just to supply this event loop.

Sometimes it is useful to be able to apply some processing within the camera event loop, that happens unconditionally to all frames. For example, an application might want to monitor the image metadata, or annotate images, all without writing an explicit loop in the application code to do this.

We would generally recommend that any such code does not take too long because, being in the middle of the camera event handling, it could easily cause frames to be dropped. It goes without saying that functions that make asynchronous requests to *Picamera2* (capturing metadata or images, for example) must be avoided as they would almost certainly lead to instant deadlocks.

There are two places where user processing may be inserted into the event loop:

- The `pre_callback`, where the processing happens before the images are supplied to applications, before they are passed to any video encoders, and before they are passed to any preview windows.

- The post_callback, where the processing happens before the images are passed to any video encoder, before they are passed to any preview windows, but *after* images have been supplied to applications.

It is not possible to do processing on frames that will be recorded as video but to avoid doing the same processing on the frames when they are displayed, or vice versa, as these two processes run in parallel. Though we note that an application could display a different stream from the one it encodes (it might display the "main" stream and encode the "lores" version), and apply processing only to one of them which would simulate this effect.

The following example uses *OpenCV* to apply a date and timestamp to every image.

```
from picamera2 import Picamera2, MappedArray
import cv2

picam2 = Picamera2()

colour = (0, 255, 0)
origin = (0, 30)
font = cv2.FONT_HERSHEY_SIMPLEX
scale = 1
thickness = 2

def apply_timestamp(request):
    timestamp = time.strftime("%Y-%m-%d %X")
    with MappedArray(request, "main") as m:
        cv2.putText(m.array, timestamp, origin, font, scale, colour, thickness)

picam2.pre_callback = apply_timestamp
picam2.start(show_preview=True)
```

Because we have used the pre_callback, it means that all images will be timestamped, whether the application requests them through any of the capture methods, whether they are being encoded and recorded as video, and indeed when they are displayed.

Had we used the post_callback instead, images acquired through the capture methods would *not* be timestamped.

Finally we draw attention to the MappedArray class. This class is provided as a convenient way to gain *in-place* access to the camera buffers - all the capture methods that applications normally use are returning copies.

The MappedArray needs to be given a request and the name of the stream for which we want access to its image buffer. It then maps that memory into user space and presents it to us as a regular *numpy* array, just as if we had obtained it via capture_array. Once we leave the with block, the memory is unmapped and everything is cleaned up.

WARNING

The amount of processing placed into the event loop should always be as limited as possible. It is recommended that any such processing is restricted to drawing in-place on the camera buffers (as above), or using metadata from the request. Above all, calls to the camera system should be avoided or handled with extreme caution as they are likely to block waiting for the event loop to complete some task - but which is now deadlocked.

Further Examples

- opencv_face_detect_3.py shows how you would draw faces on a recorded video, but not on the image used for face detection.
- timestamped_video.py writes a date/timestamp onto every frame of a recorded video.

8.2.2. Dispatching Tasks into the Event Loop

Besides using the pre and post callbacks, another way to use the event loop is to dispatch function calls to it which it will complete as and when camera images arrive. In fact, all of the "capture" type functions are implemented internally in this way.

The idea is that a list of functions can be submitted to the event loop which behaves as follows:

1. Every time a completed request is received from the camera, it calls the first function in the list.
2. The functions must always return a *tuple* of two values. The first should be a boolean, indicating whether that function is "finished". In this case, it is popped from the list, otherwise it remains at the front of the list and will be called again next time. (Note that we never move on and call the next function with the same request.)
3. If the list is now empty, that list of tasks is complete and this is signalled to the caller. The *second* value from the tuple is the one that is passed back to the caller as the result of the operation (usually through the `wait` method).

Let's look at an example.

Normally when we call `Picamera2.switch_mode_and_capture_file()`, the camera system switches from the preview to the capture mode, captures an image, then it switches back the the preview mode and starts the camera running again. What if we want to stop the camera as soon as possible after the capture? In this case we've spent time restarting the camera in the preview mode before we can call `Picamera2.stop()` from our application (and wait again for that to happen).

Here's the code:

```
from picamera2 import Picamera2

picam2 = Picamera2()
capture_config = picam2.create_still_configuration()
picam2.start()

def switch_mode_capture_file_and_stop(camera_config, file_output, name="main"):
    def capture_and_stop_(file_output):
        picam2.capture_file_(file_output, name)
        picam2.stop_()
        return (True, None)

    functions = [(lambda: picam2.switch_mode_(camera_config)),
                 (lambda: capture_and_stop_(file_output))]
    picam2.dispatch_functions(functions)
    return picam2.wait()

switch_mode_capture_file_and_stop(capture_config, "test.jpg")
```

The important points to note are:

- `switch_mode_capture_file_and_stop` creates a list of two functions that it dispatches to the event loop.
- The first of these functions (`picam2.switch_mode_`) will switch the camera into the capture mode, and then return `True` as its first value, removing it from the list.
- When the first frame in the capture mode arrives, the local `capture_and_stop_` function will run, capturing the file and stopping the camera.
- This function returns `True` as its first value as well, so it will be popped of the list. The list is now empty so the event loop will signal that it is finished.
- The `picam2.wait()` function will see this signal and return.

WARNING

Here too the application must take care what functions it calls from the event loop. For example, most of the usual *Picamera2* functions are likely to cause a deadlock. The convention has been adopted that functions that are explicitly safe should end with a `_` (an underscore).

8.3. Pixel Formats and Memory Considerations

It is in general difficult to predict which image formats will work best. Some use considerably more memory than others, and some are more widely supported than others in third party modules and libraries. Often these constraints work against one another - the most widely useful formats are the most memory hungry!

Similarly, some Pi versions have more memory available than others, and it may further depend which sensor (v1, v2 or HQ) is being used and whether full resolution images are being processed.

The table below lists the image formats that we would recommend users to choose, the size of a single full resolution 12MP image, and whether they work with certain other modules, both functionality within *Picamera2* and third party.

Table 1. Recommended image formats

	XRGB8888/XBGR8888	RGB888/BGR888	YUV420/YVU420
12MP Size	48MB	36MB	18MB
Qt GL Preview	Yes	No	Yes
Qt Preview	Yes, slow	Yes, slow	Requires OpenCV, very slow
DRM Preview	Yes	Yes	Yes
Null Preview	Yes	Yes	Yes
JPEG Encode	Yes	Yes	No
Video Encode	Yes	Yes	Yes
OpenCV	Often	Yes	Convert to RGB only

CMA Memory

CMA stands for *Contiguous Memory Allocator*, and on the Raspberry Pi it provides memory that can be used directly by the camera system and all its hardware devices. All the memory buffers for the *main*, *lores* and *raw* streams are allocated here and, being shared with the rest of the Linux Operating System, it can come under pressure and be subject to fragmentation.

When the CMA area runs out of space, this can be identified by a *Picamera2* error saying that *V4L2* (the Video for Linux kernel subsystem) has been unable to allocate buffers. Mitigations may include allocating fewer buffers (the `buffer_count` parameter in the camera configuration), choosing image formats that use less memory, or using lower resolution images. The following workarounds can also be tried.

Increase the size of the CMA area

The default CMA size on Pi devices is: 256MB if the total system memory is less than or equal to 1GB, otherwise 320MB. CMA memory is still available to the system when "regular" memory starts to run low, so increasing its size does *not* normally starve the rest of the operating system.

As a rule of thumb, all systems should usually be able to increase the size to 320MB if they are experiencing problems, 1GB systems could probably go to 384MB and 2GB or larger systems could go as far as 512MB. But you may find that different limits work best on your own systems, and some experimentation may be necessary.

To change the size of the CMA area, you will need edit the `/boot/config.txt` file. Find the line that says `dtoverlay=vc4-kms-v3d` and replace it by:

- `dtoverlay=vc4-kms-v3d,cma-320` for 320MB

- `dtoverlay=vc4-kms-v3d,cma-384` for 384MB, or
- `dtoverlay=vc4-kms-v3d,cma-512` for 512MB.

Do not add any spaces or change any of the formatting from what is given above.

NOTE

Anyone using the `fkms` driver can continue to use it and change the CMA area as described above. Just keep `fkms` in the driver name.

WARNING

Legacy camera stack users may at some point in time have increased the amount of `gpu_mem` available in their system, as this was used by the legacy camera stack. *Picamera2* and *libcamera* make no use of `gpu_mem` so we strongly recommend removing any `gpu_mem` lines from your `/boot/config.txt` as its only effect is likely to be to waste memory by making it unavailable to parts of the system that could otherwise have used it.

Working with YUV420 images

One final suggestion for reducing the size of images in memory is to use the YUV420 format instead. Third party modules often do not support this format very well, so some kind of software conversion to a more familiar RGB format may be necessary. The benefit of doing this conversion in the application is that the large RGB buffer ends up in user space where we benefit from virtual memory, and the CMA area only needs space for the smaller YUV420 version.

Fortunately, *OpenCV* provides a conversion function from YUV420 to RGB. It takes substantially less time to execute than, for example, the JPEG encoder, so for some applications it may be a good trade-off. The following example shows how to convert a YUV420 image to RGB:

```
from picamera2 import Picamera2
import cv2

picam2 = Picamera2()
picam2.create_preview_configuration({"format": "YUV420"})
picam2.start()

yuv420 = picam2.capture_array()
rgb = cv2.cvtColor(yuv420, cv2.COLOR_YUV420pRGB)
```

This function does not appear to let the application choose the YUV/RGB conversion matrix, however.

Further Examples

- yuv_to_rgb.py shows how to convert a YUV420 image to RGB using *OpenCV*.

8.4. Buffer Allocations and Queues

Related to the general issue of memory usage is the question of how we know how many buffers (the `buffer_count` parameter) we should allocate to the camera. Here are the heuristics used by *Picamera2*'s default configurations.

- Preview configurations are given 4 buffers by default. This is normally enough to keep the camera system running smoothly even when there is moderate other processing going on.
- Still capture configurations are given only 1 buffer by default. This is because they can be very large and may pose particular problems for 512MB platforms. But it does mean that, because of the way the readout from the image sensor is pipelined, we are certain to drop at least every other camera frame. If you have memory available and want fast full resolution burst captures, you may want to increase this number.

- Video configurations allocate 6 buffers. The system is likely to be more busy while recording video, so the extra buffers reduce the chances of dropping frames.

Holding on to Requests

We have seen how an application can hold on to requests for its own use, during which time they are not available to the camera system. If this causes unacceptable frame drops, or even stalls the camera system entirely, then the answer is simply to allocate more buffers to the camera at configuration time.

As a general rule, if your application will only ever be holding on to one request, then it should be sufficient to allocate just one extra buffer to restore the *status quo ante*.

Buffer Queues within *Picamera2*

Normally *Picamera2* always tries to keep hold of the most recent camera frame to arrive. For example, if an application does some processing that normally fits within a frame period, but occasionally takes a bit longer (which is always a particular risk on a multi-tasking operating system), then it's less likely to drop frames and fall behind.

The length of this queue within *Picamera2* is just a single frame. It does mean that, when you ask to capture a frame or metadata, the function is likely to return *immediately*, unless you have already made such a request just before.

The exception to this is when the camera is configured with a single buffer (the default setup for still capture), when it is not possible to hang on to the previous camera image - because there is no "spare" buffer for the camera to fill to replace it! In this case, no previous frame is held within *Picamera2*, and requests to capture a frame will *always* wait for the next one from the camera. In turn, this does mean there is some risk of image "tearing" in the preview windows (when the new image replaces the old one half way down the frame).

8.5. Using the Camera in Qt Applications

The recommended route to creating applications with a camera window embedded into the GUI is through *Qt*. In fact, *Picamera2*'s own preview windows are implemented using *Qt*, though they are somewhat unconventional and we would advise against copying them. We'll see how to make a more standard *Qt* application here.

Qt Widgets

Picamera2 provides two Qt widgets:

- `QGlPicamera2` - this is a Qt widget that renders the camera images using hardware acceleration through the Pi's GPU.
- `QPicamera2` - a software rendered but otherwise equivalent widget. This version is much slower and the `QGlPicamera2` should be preferred in nearly all circumstances except those where it does not work (for example, the application has to operate with a remote window through X forwarding).

Both widgets have an `add_overlay` method which implements the overlay functionality of *Picamera2*'s preview windows. This method accepts a single 3-dimensional *numpy* array as an RGBA image in exactly the same way, making this feature also avilable to *Qt* applications.

When the widget is created there is also an optional `keep_ar` (keep aspect ratio) parameter, defaulting to `True`. This allows the application to choose whether camera images should be letter- or pillar-boxed to fit the size of the widget (`keep_ar=True`) or stretched to fill it completely, possibly distorting the relative image width and height (`keep_ar=False`).

Finally, the two Qt widgets both support the `transform` parameter that allows camera images to be flipped horizontally and/or vertically as they are drawn to the screen (without having any effect on the camera image itself).

The Event Loop

When we're writing a *Picamera2* script that runs in the Python interpreter's main thread, the event loop that drives the camera system is supplied by the preview window (which may also be the NULL preview that doesn't display anything). In this case, however, the *Qt* event loop effectively becomes the main thread and drives the camera application too.

This is probably the simplest camera-enabled *Qt* application that we could write:

```
from PyQt5.QtWidgets import QApplication
from picamera2.previews.qt import QGlPicamera2
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration())

app = QApplication([])
qpicamera2 = QGlPicamera2(picam2, width=800, height=600, keep_ar=False)
qpicamera2.setWindowTitle("Qt Picamera2 App")

picam2.start()
qpicamera2.show()
app.exec()
```

In a real application, of course, the `qpicamera2` widget will be embedded into the layout of a more complex parent widget or window.

Invoking Camera Functions

Camera functions fall broadly into three types for the purposes of this discussion. There are:

1. Functions that return immediately and are safe to call,
2. Functions that have to wait for the camera event loop to do something before the operation is complete, so they must be called in a non-blocking manner,
3. And functions that should not be called at all.

The following table lists the public API functions and indicates which category they are in:

Table 2. Public APIs in a Qt application

Function Name	Status
`create_preview_configuration`	Safe to call directly.
`create_still_configuration`	Safe to call directly.
`create_video_configuration`	Safe to call directly.
`configure`	Safe to call directly.
`start`	Safe to call directly.
`stop`	Safe to call directly.
`start_encoder`	Safe to call directly.
`start_recording`	Safe to call directly.
`switch_mode`	Call in a non-blocking manner.
`switch_mode`	Call in a non-blocking manner.
`switch_mode`	Call in a non-blocking manner.
`switch_mode`	Call in a non-blocking manner.
`capture_file`	Call in a non-blocking manner.
`capture_buffer`	Call in a non-blocking manner.
`capture_array`	Call in a non-blocking manner.
`capture_request`	Call in a non-blocking manner.
`capture_request_and_stop`	Call in a non-blocking manner.
`capture_image`	Call in a non-blocking manner.

Function Name	Status
capture_metadata	Call in a non-blocking manner.
switch_mode_and_capture_file	Call in a non-blocking manner.
switch_mode_and_capture_buffer	Call in a non-blocking manner.
switch_mode_and_capture_array	Call in a non-blocking manner.
switch_mode_and_capture_image	Call in a non-blocking manner.
switch_mode_capture_request_and_stop	Call in a non-blocking manner.
start_and_capture_file	Do not call at all.
start_and_capture_files	Do not call at all.
start_and_record_video	Do not call at all.

Invoking a Blocking Camera Function in a Non-Blocking Manner

When we're running a script in the Python interpreter thread, the camera event loop runs asynchronously - meaning we can ask it to do things and wait for them to finish.

We accomplish this because these functions have a `wait` and a `signal_function` parameter which were discussed earlier. The default values cause the function to block until it is finished, but if we pass a `signal_function` then the call will return immediately, and we rely on the `signal_function` to return control to us.

In the *Qt* world, the camera thread and the *Qt* thread are the same, so we simply cannot block for the camera to finish - because everything will deadlock. Instead, we have to tell these functions that they *may not block*, and also provide them with an alternative *Qt*-friendly way of telling us they're finished. Therefore, both widgets provide:

- a `done_signal` member - this is a *Qt* signal to which an application can connect its own callback function and,
- a `signal_done` function - which emits the `done_signal`.

So the steps for an application are:

- Connect a callback function to the `done_signal`.
- Functions like `Picamera2.switch_mode_and_capture_file` must be given a `signal_function` to call when the operation completes (and which will cause them not to block). Normally we can simply supply the widget's `signal_done` method.
- Remember that we must still call `Picamera2.wait` when the operation has finished so as to clean everything up tidily.

We finish with a small worked example of this:

```
from PyQt5 import QtWidgets
from PyQt5.QtWidgets import QPushButton, QVBoxLayout, QApplication, QWidget

from picamera2.previews.qt import QGlPicamera2
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration())

def on_button_clicked():
    button.setEnabled(False)
    cfg = picam2.create_still_configuration()
    picam2.switch_mode_and_capture_file(cfg, "test.jpg", signal_function=qpicamera2.
signal_done)

def capture_done():
    picam2.wait()
    button.setEnabled(True)

app = QApplication([])
qpicamera2 = QGlPicamera2(picam2, width=800, height=600, keep_ar=False)
button = QPushButton("Click to capture JPEG")
window = QWidget()
qpicamera2.done_signal.connect(capture_done)
button.clicked.connect(on_button_clicked)

layout_v = QVBoxLayout()
layout_v.addWidget(qpicamera2)
layout_v.addWidget(button)
window.setWindowTitle("Qt Picamera2 App")
window.resize(640, 480)
window.setLayout(layout_v)

picam2.start()
window.show()
app.exec()
```

Observe that:

- The `capture_done` function is connected to the `done_signal`.
- When we call `switch_mode_and_capture_file` we must tell it not to block by supplying a function to call when it is finished (`qpicamera2.signal_done`).
- This function emits the `done_signal`, which gives back control in the `capture_done` function where we can re-enable the button.
- Once the operation is done, we must call the `wait()` method to complete the process before it can be run again.

And here is our sophisticated *Qt* application in action:

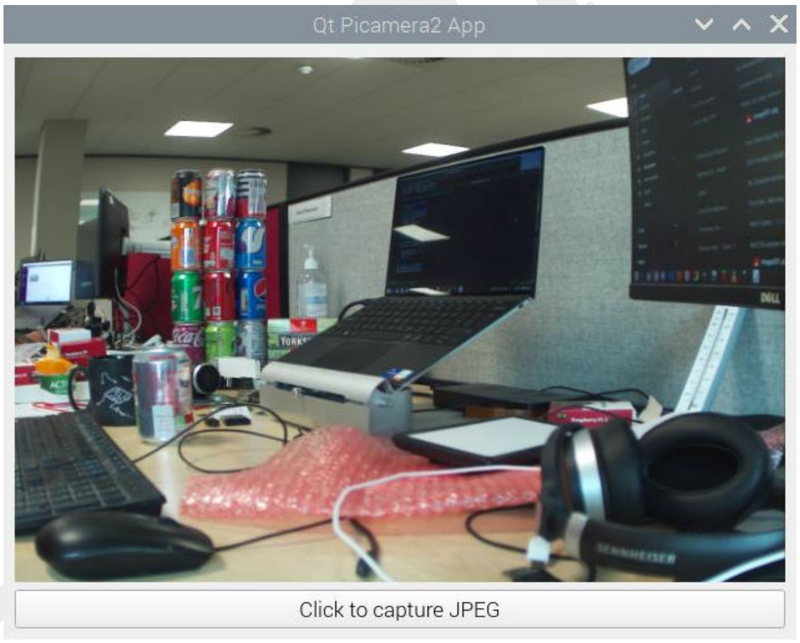

Figure 4. A Qt app using Picamera2

Further Examples

- app_capture2.py lets you capture a JPEG by clicking a button, and then re-enables the button afterwards using the *Qt* signal mechanism.
- app_capture_overlay.py demonstrates the use of overlays in the *Qt* widgets. (app_recording.py shows you how to record a video from a *Qt* application.

Chapter 9. Application Notes

This section answers a selection of "how to" questions for particular use-cases.

9.1. Streaming to a Network

There are some streaming examples available as follows:

- capture_stream.py - streaming to a TCP socket.
- capture_stream_udp.py - streaming to a UDP socket.
- mjpeg_server.py - a simple webserver to stream MJPEG video to a web page.

There are further examples below showing how to send HSL or MPEG-DASH live streams, and also one where we send an MPEG-2 Transport Stream to a socket.

9.2. Output using FFmpeg

The `FfmpegOutput` class has been dicussed previously. Here we give some more complex examples.

9.2.1. HLS Live Stream

The following `FfmpegOutput` example shows how to generate an HLS live stream:

```
from picamera2.outputs import FfmpegOutput

output = FfmpegOutput("-f hls -hls_time 4 -hls_list_size 5 -hls_flags delete_segments
-hls_allow_cache 0 stream.m3u8")
```

You would also have to start an HTTP server to enable remote clients to access the stream. One simple way to do this is to open another terminal window in the same folder and enter

```
python3 -m http.server
```

9.2.2. MPEG-DASH Live Stream

This `FfmpegOutput` example shows how to generate an MPEG-DASH live stream:

```
from picamera2.outputs import FfmpegOutput

output = FfmpegOutput("-f dash -window_size 5 -use_template 1 -use_timeline 1 stream.mpd")
```

You would also have to run a HTTP server just as we saw previously:

```
python3 -m http.server
```

9.2.3. Sending an MPEG-2 Transport Stream to a Socket

We can create an MPEG-2 Transport Stream and stream it to a UDP socket as follows. We have to specify the format `-f mpegts` because the rest of the output name does not allow it to be deduced:

```
from picamera2.outputs import FfmpegOutput

output = FfmpegOutput("-f mpegts udp://<ip-addr>:<port>")
```

Adding `audio=True` will add and send an audio stream if a microphone is available.

9.3. Multiple Outputs

When recording video, it is possible to send the output to more than one place simultaneously. The `FileOutput` can be started and stopped explicitly by the application at any time. All we have to do is set its `fileoutput` property and call the `start()` method.

In this example we stream an MPEG-2 Transport Stream over the network using the UDP protocol where any client may connect and view the stream. After 5 seconds we start the second output and record 5 seconds worth of H.264 video to a file. We close this output file, but the network stream continues to play.

```
from picamera2 import Picamera2
from picamera2.encoders import H264Encoder
from picamera2.outputs import FileOutput, FfmpegOutput
import time

picam2 = Picamera2()
video_config = picam2.create_video_configuration()
picam2.configure(video_config)

encoder = H264Encoder(repeat=True, iperiod=15)
output1 = FfmpegOutput("-f mpegts udp://<ip-address>:12345")
output2 = FileOutput()
encoder.output = [output1, output2]

# Start streaming to the network.
picam2.start_encoder(encoder)
picam2.start()
time.sleep(5)

# Start recording to a file.
output2.fileoutput = "test.h264"
output2.start()
time.sleep(5)
output2.stop()

# The file is closed, but carry on streaming to the network.
time.sleep(9999999)
```

9.4. Manipulate Camera Buffers in Place

We have already presented one example where we manipulate camera buffers in place.

Here, we imagine that we want to receive a camera image and use *OpenCV* to perform face detection. Rather than using *OpenCV* to display the resulting image (which would be at the framerate we can perform the face detection), we are going to use *Picamera2*'s own preview window, running at 30 frames per second.

Before each frame is displayed, we draw the face rectangles in place onto the image. The face rectangles correspond to the locations returned when the face detector last ran - so while the preview image updates at the full rate, the face boxes move only at the face detector’s rate.

```
from picamera2 import Picamera2, MappedArray
import cv2

face_detector = cv2.CascadeClassifier("/path/to/haarcascade_frontalface_default.xml")

def draw_faces(request):
    with MappedArray(request, "main") as m:
        for f in faces:
            (x, y, w, h) = [c * n // d for c, n, d in zip(f, (w0, h0) * 2, (w1, h1) * 2)]
            cv2.rectangle(m.array, (x, y), (x + w, y + h), (0, 255, 0, 0))

picam2 = Picamera2()
config = picam2.create_preview_configuration(main={"size": (640, 480)},
                                              lores={"size": (320, 240), "format": "YUV420"})
picam2.configure(config)

(w0, h0) = picam2.stream_configuration("main")["size"]
(w1, h1) = picam2.stream_configuration("lores")["size"]
faces = []
picam2.post_callback = draw_faces

picam2.start(show_preview=True)
while True:
    array = picam2.capture_array("lores")
    grey = array[h1,:]
    faces = face_detector.detectMultiScale(grey, 1.1, 3)
```

Appendix A: Pixel and Image Formats

The table below lists the image and pixel formats supported by *Picamera2*. For each format we list:

- The number of bits per pixel.
- The optimal alignment for this format in units of pixels.
- The shape of an image as reported by *numpy* on an array obtained using `capture_array` (where supported).

Table 3. Different image formats

	Bits per pixel	Optimal Alignment	Shape	Description
XBGR8888	32	16	(*height*, *width*, 4)	RGB format with an alpha channel. Each pixel is laid out as `[R, G, B, A]` where the `A` (or alpha) value is fixed at 255.
XRGB8888	32	16	(*height*, *width*, 4)	RGB format with an alpha channel. Each pixel is laid out as `[B, G, R, A]` where the `A` (or alpha) value is fixed at 255.
BGR888	24	32	(*height*, *width*, 3)	RGB format. Each pixel is laid out as `[R, G, B]`.
RGB888	24	32	(*height*, *width*, 3)	RGB format. Each pixel is laid out as `[B, G, R]`.
YUV420	12	64	(*height**3/2, *width*)	YUV 4:2:0 format. There are *height* rows of Y values, then *height*/2 rows of half-width U and *height*/2 rows of half-width V. The array form has 2 rows of U (or V) values on each row of the matrix.
YVU420	12	64	(*height**3/2, *width*)	YUV 4:2:0 format. There are *height* rows of Y values, then *height*/2 rows of half-width V and *height*/2 rows of half-width U. The array form has 2 rows of V (or U) values on each row of the matrix.
NV12	12	32	Unsupported	YUV 4:2:0 format. A plane of *height*stride** Y values followed by *height*/2**stride* interleaved U and V values.
NV21	12	32	Unsupported	YUV 4:2:0 format. A plane of *height*stride* Y values followed by *height*/2**stride* interleaved V and U values.
YUYV	16	32	Unsupported	YUV 4:2:2 format. A plane of *height*stride* interleaved values in the order Y U Y V for every 2 pixels.
YVYU	16	32	Unsupported	YUV 4:2:2 format. A plane of *height*stride* interleaved values in the order Y V Y U for every 2 pixels.
UYVY	16	32	Unsupported	YUV 4:2:2 format. A plane of *height*stride* interleaved values in the order U Y V Y for every 2 pixels.
VYUY	16	32	Unsupported	YUV 4:2:2 format. A plane of *height*stride* interleaved values in the order V Y U Y for every 2 pixels.

The final table lists the extent of support for each of these formats, both in *Picamera2* and in some third party libraries.

Table 4. Support for different image formats

	XRGB8888/XBGR8888	RGB888/BGR888	YUV420/YVU420	NV12/NV21	YUYV/UYVY	YVYU/VYUY
Capture Buffer	Yes	Yes	Yes	Yes	Yes	Yes
Capture Array	Yes	Yes	Yes	No	No	No
Qt GL Preview	Yes	No	Yes	No	Yes	No
Qt Preview	Yes, slow	Yes, slow	Requires OpenCV, very slow	No	No	No

	XRGB8888/XBGR8888	RGB888/BGR888	YUV420/YVU420	NV12/NV21	YUYV/UYVY	YVYU/VYUY
DRM Preview	Yes	Yes	Yes	No	No	No
Null Preview	Yes	Yes	Yes	Yes	Yes	Yes
JPEG Encode	Yes	Yes	No	No	No	No
Video Encode	Yes	Yes	Yes	Yes	Yes	Yes
OpenCV	Often	Yes	Convert to RGB only	No	No	No
Python Image Library	Yes	Yes	No	No	No	No

Appendix B: Camera Configuration Parameters

The table below lists all camera configuration parameters, the allowed values, and gives a description. We start with the global parameters, that is, ones that are not specific to any of the *main*, *lores* or *raw* streams.

Table 5. Global camera configuration parameters

Parameter name	Permitted values	Description

`"use_case"`	`"preview"`
	`"still"`
	`"video"`
		This parameter only exists as an aid to users, so as to show what the intented use for a configuration was. The `create_preview_configuration()` will create configurations where this is set to `"preview"`, and similarly for the still and video versions of this method.

`"transform"`	`Transform()`
	`Transform(hflip=1)`
	`Transform(vflip=1)`
	`Transform(hflip=1, vflip=1)`
		The 2-d plane transform that is applied to all images from all the configured streams. The listed values represent, respecitvely, the identity transform, a horizontal mirror, a vertical flip and a 180 degree rotation. The default is always the identity transform.

`"colour_space"`	`Jpeg()`
	`Smpte170m()`
	`Rec709()`
		The colour space to be used for the *main* and *lores* streams. The allowed values are either the JPEG colour space (meaning sRGB primaries and transfer function and full-range BT.601 YCbCr encoding), the SMPTE 170M colour space or the Rec.709 colour space.
		
		`create_preview_configuration()` and `create_still_configuration()` both default to `Jpeg()`. `create_video_configuration()` chooses `Jpeg()` if the *main* stream is using an RGB format. For YUV formats it will default to `Smpte170m()` if the resolution is smaller than 1280x720, otherwise `Rec709()`.
		
		For any *raw* stream, the colour space will always implicitly be the image sensor's native colour space.

`"buffer_count"`	`1`, `2`, ...	The number of sets of buffers to allocate for requests, which becomes the number of "request" objects that are available to the camera system. By default we choose 4 for preview configurations, 1 for still capture configurations, and 6 for video. Increasing the number of buffers will tend to lead to fewer frame drops, though with diminishing returns. The maximum possible number of buffers depends on the platform, the image resolution, the amount of CMA allocated.

`"display"`	`None`
	`"main"`
	`"lores"`
		The name of the stream that will be displayed in the preview window (if one is running). Normally the *main* stream will be displayed, though the *lores* stream can be shown instead when it is defined. By default, `create_still_configuration()` will use the value `None`, as the buffers are typically very large and can lead to memory fragmentation problems in some circumstances if the display stack is holding on to them.

`"encode"`	`None`
	`"main"`
	`"lores"`
		The name of the stream that will be used for video recording. By default `create_video_configuration()` will set this to the *main* stream, thought the *lores* stream can also be used if it is defined. For preview and still use cases the value will be set to `None`.

Parameter name	Permitted values	Description
`"controls"`	Please refer to the camera controls section.	With this parameter we can specify a set of runtime controls that can be regarded as part of the camera configuration, and applied whenever the configuration is (re-)applied. Different use cases may also supply some slightly different default control values.

Next we list the stream-specific configuration parameters.

Table 6. Stream-specific configuration parameters

Parameter name	Permitted values	Description
`"format"`	A string describing the image format. Please refer to this table.	The pixel and image format. Please refer to the linked table for a full description. For *raw* streams the permitted values will be raw image formats and can be listed either by querying the `Picamera2.sensor_modes` property or from a terminal by running `libcamera-hello --list-cameras`. The formats will be the strings that start with an uppercase `S` and are followed by something like `RGGB` or `GBRG`. The trailing `_CSI2P` may be omitted if unpacked raw pixels are required, though in most cases this it not recommended as it uses more total memory and more system memory bandwidth.
`"size"`	`(width, height)`	A tuple of two values giving the width and height of the output image. Both numbers should be no less than 64. For raw streams, the allowed resolutions are listed again by `libcamera-hello --list-cameras`, along with the correct format to use for that resolution. You can pick different sizes, but the system will simply use whichever of the allowed values it deems to be "closest".

Appendix C: Camera Controls

Exhaustive list and description of all camera controls, including all the minimum and maximum values.

Appendix D: Camera Properties

Exhaustive list and description of all camera properties. What they mean, which ones can change after re-configuring.