The Picamera2 Library

A libcamera-based Python library for Raspberry Pi cameras

Colophon

Release	3
Build date	30/06/2026
Build version	ca0fec80aab2

Legal disclaimer notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI LTD ("RPL") "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

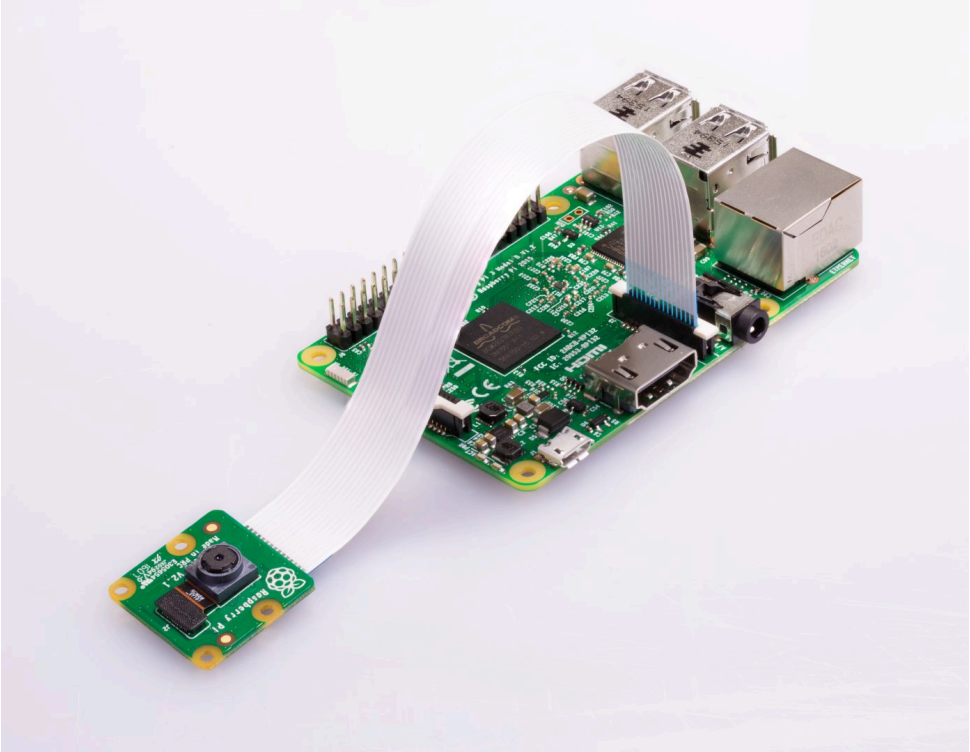
Raspberry Pi products are provided subject to RPL's Standard Terms. RPL's provision of the RESOURCES does not expand or otherwise modify RPL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Table of Contents

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 not for other types of camera, although there is some limited support for USB cameras.

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. 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.

Picamera2 is the replacement for the legacy PiCamera Python library. It provides broadly the same facilities, although many of these capabilities are exposed differently. Picamera2 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 the legacy camera stack should continue to use the old PiCamera library. Note that the legacy camera stack and the old PiCamera library have been deprecated for a number of years and no longer receive any kind of support.

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 two-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.

1.0.1. Software version

This manual describes *Picamera2* version 0.3.26 which is the latest release at the time of writing.

2. Getting Started

2.1. Requirements

Picamera2 is designed for systems running either Raspberry Pi OS or Raspberry Pi OS Lite, using a Bullseye or later image. Picamera2 is pre-installed in current images obtained using the Raspberry Pi Imager tool. Alternatively the latest images can also be downloaded from the Raspberry Pi website. We strongly recommend users with older images to consider updating them or to proceed to installation instructions in Section 2.2. Installation and updating.

Picamera2 can operate in a headless manner, not requiring an attached screen or keyboard. 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 and later images by default run the libcamera camera stack, which is required for Picamera2. You can check that libcamera is working by opening a command window and typing:

```
Code
rpicam-hello
```

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

2.1.1. Using lower-powered devices

Some lower-powered devices, such as the Raspberry Pi Zero, are generally much slower at running desktop GUI (Graphical User Interface) software. Correspondingly, performance may be poor trying to run the camera system with a preview window that has to display images through the GUI's display stack.

On such devices we would recommend either not displaying the images, or displaying them without the GUI. The Pi can be configured to boot to the console (avoiding the GUI) using the `raspi-config` tool, or if you are using the GUI 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

We strongly recommend starting by updating to the latest version of Raspberry Pi OS. Most straightforwardly, this can be downloaded using the Raspberry Pi Imager tool for flashing micro SD cards. We also recommend updating your Raspberry Pi installation once it's booted with

```
Code
sudo apt update
sudo apt full-upgrade
```

before proceeding further.

As of mid-September 2022, Picamera2 is pre-installed in Raspberry Pi OS images, but not in Raspberry Pi OS Lite images. Where Picamera2 is not pre-installed, it can be installed with

```
Code
sudo apt install -y python3-picamera2 --no-install-recommends
```

which installs a slightly reduced version with fewer windowing or GUI components (this would be suitable for a "Lite" system). Or to install a full version, use

```
Code
sudo apt install -y python3-picamera2
```

When updating your Picamera2 installation we recommend doing so through a full system upgrade as we did earlier (`sudo apt update` followed by `sudo apt full-upgrade`). This ensures that all your system libraries are guaranteed to be compatible. Note that you should always back up anything critical from your system before performing the full upgrade.

2.2.1. Installation using pip

Warning

We strongly recommend installing and updating Picamera2 using `apt` which will avoid compatibility problems. If you do wish to install Picamera2 using `pip`, please read to the end of this section before starting. On Bookworm and later images you will also need to be familiar with Python virtual environments.

The easiest way to get the dependencies that you need will be to install Picamera2 first using `apt` if it was not already pre-installed (see above). Subsequently, any virtual environment should be created to inherit system packages (for example, use the `--system-site-packages` option when creating a *venv*). A subsequent `pip` installation will then take precedence over the one from `apt`:

Code

```
pip install picamera2
```

This does rely on the new version of Picamera2 being compatible with the libcamera Python bindings that were installed from `apt`. Note that there is **no** libcamera package available through `pip`, so if you find that your version of Picamera2 is not compatible then you may need to build libcamera for yourself.

Users wanting to use Picamera2 with different operating systems that do not include libcamera, or with different non-system versions of Python, are also likely to need to build their own version of libcamera. Instructions for this are available on the Raspberry Pi website, however, it does lie beyond the scope of this guide.

2.2.2. Configuring the `/boot/firmware/config.txt` file

Normally no changes should be required to the `/boot/firmware/config.txt` file (or `/boot/config.txt` on some old versions of the OS) 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 two seconds and capture a JPEG file (still in the preview resolution)

Note

Users of Raspberry 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.

GUI users should enter:

Code

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

```
Code
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 and "just want a picture" (or video), though most users will want a more complete understanding of what happens under the hood, as we saw Section 2.3. A first example above.

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

```
Code
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 five second video:

```
Code
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. Multiple Cameras

With the appropriate hardware, it is possible to connect multiple cameras to a Raspberry Pi and, with a compute module or a Raspberry Pi 5, to drive two of them simultaneously. For more information, please refer to Section 8.7. Multiple Cameras.

2.6. Additional software

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

2.6.1. OpenCV

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

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

Installing from `apt` also ensures you do not get a version that is incompatible with the *Qt* GUI toolkit that is installed with Picamera2.

2.6.2. TensorFlow Lite

TensorFlow Lite can be installed with:

```
Code
pip3 install tflite-runtime
```

2.6.3. 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:

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

2.7. Further examples

Throughout this guide we'll give lots of examples, but we'll also highlight some of those 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.

3. Preview Windows

3.1. Preview window parameters

In Section 2.3. A first example we have already seen two different types of preview window. There are in fact four different versions, which Section 3.2. Preview window implementations 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:

```
Code
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 in Section 4.2.1.1. More on transforms.

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 the GUI environment and we would recommend it in nearly all circumstances when a GUI is required.

The QtGL preview window can be started with:

```
Code
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 Section 3.2.3. Qt preview.

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 Raspberry Pi 4 and later devices 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:

```
Code
from picamera2 import Picamera2, Preview

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

Because X Windows is not running, it is not possible to move or resize this window with the mouse.

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.

Note

The DRM/KMS preview window is **not** supported when using the legacy `fkms` display driver. Please use the recommended `kms` display driver (`dtoverlay=vc4-kms-v3d` in your `/boot/config.txt` file) instead.

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 computationally costly and should be avoided where possible. Even a Raspberry Pi 4 will start to struggle once the preview window size increases.

The Qt preview can be started with:

```
Code
from picamera2 import Picamera2, Preview

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

The main use case for the Qt preview is displaying the preview window on 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 Raspberry 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 can start the NULL preview explicitly like this:

```
Code
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 three 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 be stopped; an alternative one should then be started. 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.

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. 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:

```
Code
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. All the buffers that are available will be filled with camera images by libcamera. 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 your Raspberry 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, the GUI 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 appear on the display that is attached directly to the Pi.

3.5. Other Preview Features

3.5.1. Setting the Preview Title Bar

For the Qt and QtGL preview windows, camera information may optionally be displayed on the window title bar. Specifically, information from the metadata that accompanies each image can be displayed by listing the required metadata fields in the *Picamera2* object's `title_fields` property. For example

```
Code
from picamera2 import Picamera2

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

picam2.title_fields = ["ExposureTime", "AnalogueGain"]
```

This will display every image's exposure time and analogue gain values on the title bar. If any of the given field names are misspelled or unavailable then the value `INVALID` is reported for them.

When using the NULL preview or DRM preview, or when *Picamera2* is embedded in a larger *Qt* application, then the `title_fields` property has no effect.

The metadata that is available can easily be inspected using the `capture_metadata` method. Alternatively, more information on the different forms of metadata is available in Appendix C. Camera controls.

3.5.2. Further Preview Topics

Further preview features are covered 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 in Section 8.5. Using the camera in Qt applications.

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

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 the configuration must be applied to the camera system (using the `Picamera2.configure` method), and that 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

There is nothing inherently different about any one of these methods over another, they differ only in that they supply slightly different defaults so that it's easier to get something appropriate to the use case at hand. But, if you choose the necessary parameters carefully, you can use a configuration from any of these functions for any use case.

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

```
Code
from picamera2 import Picamera2

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. Additionally, once a configuration object has been created, applications are free to alter the object's recommendations before calling `picam2.configure`.

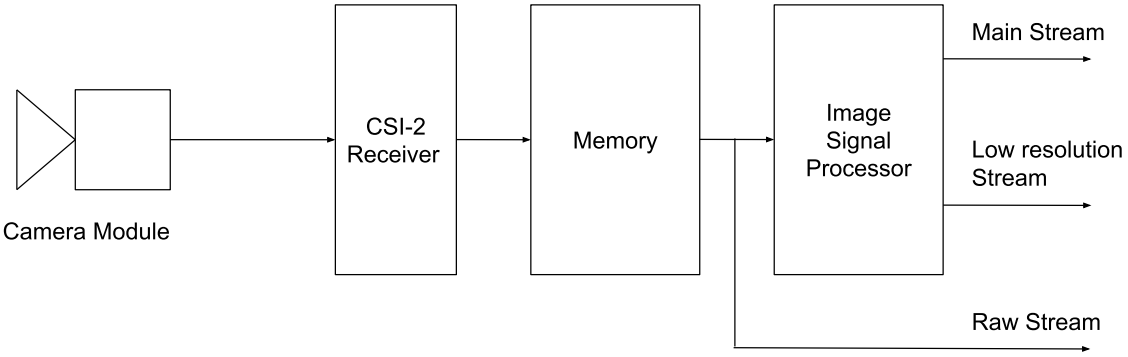
One thing we shall learn is that configurations are just Python dictionaries, and it's easy for us to inspect them and see what they are saying.

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 Raspberry 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 (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; it must be no larger than the main image. On a Pi 4 or earlier device, the lores image *must* be in a YUV format, whereas on a Pi 5 (or later device) it can be RGB or YUV, like 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.
- Some applications need to be able to control the mode (resolution, bit depth and so on) that the sensor is running in. This can be done using the *sensor* part of the camera configuration or, if this is absent, it will be inferred from the specification of the raw stream (if present).
- 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.
- `queue` - whether the system is allowed to queue up a frame ready for a capture request.

- `sensor` - parameters that allow an application to select a particular mode of operation for the sensor. This is quite an involved topic, which we shall cover in Section 4.2.2.3. Raw streams and the Sensor Configuration later.
- `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. This parameter only specifies a default value that can be overridden when the encoders are started.

More on transforms

Transforms can be constructed as shown below:

```
Code
> 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:

```
Code
from picamera2 import Picamera2
from libcamera import Transform

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

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

More on colour spaces

The implementation of colour spaces in libcamera 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).

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

```
Code
> from libcamera import ColorSpace
> ColorSpace.Sycc()
<libcamera.ColorSpace 'sYCC'>
> ColorSpace.Smpte170m()
<libcamera.ColorSpace 'SMPTE170M'>
> 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:

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

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

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

- `create_preview_configuration` and `create_still_configuration` will use the sYCC 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 sYCC 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.

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 four sets of buffers
- `create_still_configuration` requests just one set of buffers (as these are normally large full resolution buffers)
- `create_video_configuration` requests six 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:

```
Code
from picamera2 import Picamera2

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

More on the queue parameter

By default, *Picamera2* keeps hold of the last frame to be received from the camera and, when you make a capture request, this frame may be returned to you. This can be useful for burst captures, particularly when an application is doing some processing that can take slightly longer than a frame period. In these cases, the queued frame can be returned immediately rather than remaining idle until the next camera frame arrives.

But this does mean that the returned frame can come from slightly before the moment of the capture request, by up to a frame period. If this behaviour is not wanted, please set the `queue` parameter to `False`. For example:

```
Code
from picamera2 import Picamera2

picam2 = Picamera2()
preview_config = picam2.create_preview_configuration(queue=False)
```

Note that, when the `buffer_count` is set to one, as is the case by default for still capture configurations, then no frames are ever queued up (because holding on to the only buffer would completely stall the camera pipeline).

More on the sensor parameters

These work in conjunction with some of the stream parameters and are discussed in Section 4.2.2.3. Raw streams and the Sensor Configuration.

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:

```
Code
from picamera2 import Picamera2

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

This would request 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.

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:

```
Code
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.

NOTE [The `encode` parameter in the configuration is retained principally for backwards compatibility. Recent versions of Picamera2 allow multiple encoders to run at the same time, using either the same or different streams. For this reason, the `Picamera2.start_encoder` and `Picamera2.start_recording` methods accpet a `name` parameter to define which stream is being recorded. If this is omitted, the `encode` stream from the configuration will be used.]

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 stream is not normally delivered to applications unless it was requested.

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 by default values:

```
Code
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, Padding and Alignment

The configuration-generating functions can make minimal changes to the configuration where they detect something is invalid, but otherwise they will attempt to adhere to the values given. In previous versions of Picamera2, some image sizes led to more efficient processing than others; this is no longer the case. Nevertheless the width alignment of images may still be a consideration when using YUV420 format images.

The image processing hardware on the Pi cannot start writing a row of pixels to an arbitrary memory location - normally it will have to be (for example) a 64-byte aligned address, meaning that rows of pixels may have "padding" at the end so that the next row starts at an appropriate address. An application is never usually aware of this, because the "padding" is sliced off at essentially zero cost.

However, with YUV420 images this isn't the case. Here, the rows in the U and V planes will have padding in different places to the Y rows, so removing the padding would require copying the data. Instead, the image arrays returned via (for example) `capture_array` *still contain the padding*. This is almost entirely transparent to users, as still images and videos can be captured and encoded correctly (ignoring the padding) in the usual manner. However, if you are going to do your own custom processing on these images, and cannot easily ignore the padding, then you may want to consider "aligning" the image size so as to avoid the padding entirely.

If a user really needs an image with no padding, they can use the `align_configuration` method. For example:

```
Code
> from picamera2 import Picamera2
```

```
> picam2 = Picamera2()
> config = picam2.create_preview_configuration({"size": (800, 600), "format": "YUV420"})
> config["main"]
{'format': 'YUV420', 'size': (800, 600)}
> picam2.align_configuration(config)
# Picamera2 has decided a 768x600 image will have no padding.
> config["main"]
{'format': 'YUV420', 'size': (768, 600)}
> picam2.configure(config)
> config["main"]
{'format': 'YUV420', 'size': (768, 600), 'stride': 768, 'framesize': 691200}
```

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, which is now the same as the width in pixels), and the total amount of memory every such image will occupy.

Image Formats

A wide variety of image formats are supported by libcamera, as described in Figure 5. 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. (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 on Pi 4 and earlier devices. On Pi 5, the lores stream may specify an RGB format.

Warning

Picamera2 takes its pixel format naming from libcamera, which in turn takes them from certain underlying Linux components. The results are not always the most intuitive. For example, OpenCV users will typically want each pixel to be a (B, G, R) triple for which the `RGB888` format should be chosen, and *not* `BGR888`. Similarly, OpenCV users wanting an alpha channel should select `XRGB8888`.

Raw streams and the Sensor Configuration

Image sensors normally have a number of different *modes* in which they can operate. Modes are distinguished by producing different output resolutions, some of them may give you a different field of view, and there's often a trade-off where lower resolution sensor modes can produce higher framerates. In many applications, it's important to understand the available sensor modes, and to be able to choose the one that is the most appropriate.

We can inspect the available sensor modes by querying the `sensor_modes` property of the `Picamera2` object. You should normally do this as soon as you open the camera, because finding out all the reported information will require the camera to be stopped, and will reconfigure it multiple times. For the HQ camera, we obtain:

Code

```
> 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). We recommend anyone wanting to access the raw pixel data to ask for the unpacked version of the format.

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).

Note

For a raw stream, the format normally begins with an `S`, followed by four 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 eight-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 ten-bit pixel values will be stored in every five bytes, or two twelve-bit pixel values in every three bytes. When `_CSI2P` is absent, it means the pixels will each be unpacked into a 16-bit word (or eight-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. Pi 5 uses a compressed raw format in place of the `_CSI2P` version, and which we shall learn about later.

The Sensor Configuration

Warning

This discussion of the sensor configuration applies only to Raspberry Pi OS Bookworm or later. Bullseye users should configure the sensor by specifying the raw stream format, as shown here, except that the `'sensor'` field will not be reported back as part of the applied configuration. Both packed (format ending in `_CSI2P`) and unpacked formats may be requested. Configuring the raw stream in this way is also supported in Bookworm, for backwards compatibility with Bullseye.

The best way to ask for a particular sensor mode is to set with `sensor` parameter when requesting a camera configuration. The sensor configuration accepts the following parameters:

- `output_size` - the resolution of the sensor mode, which you can find in the `size` field in the sensor modes list, and
- `bit_depth` - the bit depth of each pixel sample from the sensor, also available in the sensor modes list.

All other properties are ignored. So for example, to request the fast framerate sensor mode (the first in the list) we could generate a configuration like this:

```
Code
mode = picam2.sensor_modes[0]
config = picam2.create_preview_configuration(sensor={'output_size': mode['size'], 'bit_depth': mode['bit_depth']})
picam2.configure(config)
```

Whatever output size and bit depth you specify, Picamera2 will try to choose the best match for you. In order to be sure you get the sensor mode that you want, specify the exact size and bit depth for that sensor mode.

The Raw Stream Configuration

On a Pi 4 (or earlier)

The raw stream configuration will always be filled in for you from the sensor configuration where a sensor configuration was given. For example:

```
Code
> modes = picam2.sensor_modes
> mode = modes[0]
> mode
{'format': SRGGB10_CSI2P, 'unpacked': 'SRGGB10', 'bit_depth': 10, 'size': (1332, 990), 'fps': 120.05, 'crop_limits':
(696, 528, 2664, 1980), 'exposure_limits': (31, 2147483647, None)}
> config = picam2.create_preview_configuration(sensor={'output_size': mode['size'], 'bit_depth': mode['bit_depth']})
> picam2.configure(config)
> picam2.camera_configuration()['raw']
{'format': 'SBGGR10_CSI2P', 'size': (1332, 990), 'stride': 1696, 'framesize': 1679040}
```

Note how the raw stream configuration has been updated to match the sensor configuration even though we never specified it. By default we get packed raw pixels; had we wanted unpacked pixels it would have been sufficient to request an unpacked raw format (even though the size, bit depth and Bayer order may get overwritten):

```
Code
> config = picam2.create_preview_configuration(raw={'format': 'SRGGB12'}, sensor={'output_size': mode['size'],
'bit_depth': mode['bit_depth']})
> picam2.configure(config)
> picam2.camera_configuration()['raw']
{'format': 'SBGGR10', 'size': (1332, 990), 'stride': 2688, 'framesize': 2661120}
```

Again, the raw stream has been updated and the request for unpacked pixels respected, even though the bit depth in the raw format has been changed to match the value in the sensor configuration.

For backwards compatibility with earlier versions of Picamera2, when no sensor configuration is given but a raw stream configuration *is* supplied, Picamera2 will take the size and bit depth from the raw stream in order to select the correct sensor configuration. After Picamera2 is configured, the configuration that you supplied will have the updated sensor configuration filled in for you. For example:

```
Code
> config = picam2.create_preview_configuration(raw={'format': 'SRGGB10', 'size': (1332, 990)})
> picam2.configure(config)
> picam2.camera_configuration()['sensor']
{'bit_depth': 10, 'output_size': (1332, 990)}
```

On a Pi 5 (or later)

The situation on a Pi 5 is very similar, so familiarity with the Pi 4 discussion above will be assumed.

- The sensor configuration is the best way to request a particular sensor mode.
- When present, the sensor configuration takes precedence.
- A raw stream with unpacked pixels can be requested by asking for a raw stream with an unpacked format, with everything else coming from the sensor configuration.
- For backwards compatibility, if no sensor configuration is specified, the raw stream (if present) will be used instead.

However, the Pi 5 is different because it uses different raw stream formats. It supports:

- Compressed raw pixels. Here, camera samples are compressed to 8 bits per pixel using a visually lossless compression scheme. This format is more memory efficient, but not suitable if an application wants access to the raw pixels (because they would have to be decompressed).
- Uncompressed pixels. These are stored as one pixel per 16-bit word. It differs from the Pi 4 unpacked scheme in that pixels are *left-shifted* so that the zero padding bits are at the least significant end (the earlier devices will pad with zeros at the *most* significant end), and the full 16-bit dynamic range is used. These pixels have never been through the compression scheme, and so give a bit-exact version of what came out of the sensor (allowing for the shift that is applied).

Again for backwards compatibility, Picamera2 will translate a request for *packed* pixels in the raw stream into a request for *compressed* pixels on a Pi 5. Requests for *unpacked* raw stream pixels will be translated into *uncompressed* (and left-shifted) pixels.

Let's see this in action.

Code

```
> config = picam2.create_preview_configuration(raw={'format': 'SBGGR10_CSI2P', 'size': (1332, 990)})
> picam2.configure(config)
> picam2.camera_configuration()['sensor']
{'bit_depth': 10, 'output_size': (1332, 990)}
> picam2.camera_configuration()['raw']
{'format': 'BGGR16_PISP_COMP1', 'size': (1332, 990), 'stride': 1344, 'framesize': 1330560}
```

Here, the 'SBGGR10_CSI2P' format, which would have remained unchanged on a Pi 4, has become 'BGGR16_PISP_COMP1' or "BGGR order 16-bit Pi ISP Compressed 1" format. And the sensor configuration tells us the sensor mode, as it did before.

Looking at the unpacked/uncompressed case:

Code

```
> config = picam2.create_preview_configuration(raw={'format': 'SBGGR10', 'size': (1332, 990)})
> picam2.configure(config)
> picam2.camera_configuration()['sensor']
{'bit_depth': 10, 'output_size': (1332, 990)}
> picam2.camera_configuration()['raw']
{'format': 'SBGGR16', 'size': (1332, 990), 'stride': 2688, 'framesize': 2661120}
```

The sensor configuration tells us the sensor mode as it did in the previous example, but the 'SBGGR10' format that would have remained unchanged on a Pi 4 has become 'SBGGR16' (BGGR order 16-bits per pixel, full 16-bit dynamic range). It should be clear that, in order to determine the bit depth that the sensor is operating in, an application needs to check the *sensor configuration*, and that this method works on all Pi devices. Checking the raw stream format for this value will not work on a Pi 5 or later device.

Tip

After configuring the camera, it's often helpful to inspect `picam2.camera_configuration()` to check what you actually got!

4.2.3. Configurations and runtime camera controls

The final element in a Picamera2 configuration is runtime camera controls. As the description suggests, we normally apply 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:

Code

```
> 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 a 25 frames-per-second video you could use:

Code

```
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 Appendix C. Camera controls.

4.2.4. 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 `SensorConfiguration` object for setting the sensor mode
- 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

Code

```
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

Code

```
picam2.create_preview_configuration({"size": (800, 600)})
```

As another piece of shorthand, the `main` field can be elided, so

```
Code
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:

```
Code
configuration_object.enable_lores()
```

or

```
Code
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:

```
Code
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.

Sensor configuration and the raw stream

The sensor configuration and raw stream behave in the same way as they do when using dictionaries instead of configuration objects. This means that:

- When the `sensor` object in the camera configuration contains proper values, they will take precedence over any in the raw stream.
- When the `sensor` object does not contain meaningful values, any values in the raw stream will be used to decide the sensor mode.
- After configuration, the `sensor` field is updated to reflect the sensor mode being used, whether or not it was configured previously.

The `SensorConfiguration` object allows the application to set the `output_size` and `bit_depth`, just as was the case with dictionaries. For example on a Pi 4:

```
Code
> picam2 = Picamera2()
> picam2.preview_configuration.sensor.output_size = (1332, 990)
> picam2.preview_configuration.sensor.bit_depth = 10
> picam2.configure('preview')
> picam2.preview_configuration.raw
StreamConfiguration({'size': (1332, 990), 'format': 'SBGGR10_CSI2P', 'stride': 1696, 'framesize': 1679040})
```

Here, the raw stream has been configured from the programmed sensor configuration. But if we don't fill in the sensor configuration, it will be deduced from the raw stream (for backwards compatibility).

```
Code
> picam2 = Picamera2()
> picam2.preview_configuration.raw.size = (1332, 990)
> picam2.preview_configuration.raw.format = 'SBGGR10'
> picam2.configure('preview')
> picam2.preview_configuration.sensor
SensorConfiguration({'output_size': (1332, 990), 'bit_depth': 10})
```

Warning

This means that, because these configuration objects are persistent, after doing one configuration you can no longer update just the raw stream and expect a different sensor mode to be chosen. The application should either update the sensor configuration, or, if it wants to deduce it from the raw stream again, the sensor configuration should be cleared: `picam2.preview_configuration.sensor = None`

Stream alignment

Just as with the dictionary method, there is normally no need to align stream sizes in any particular way, but it can be accomplished using the configuration object's `align` method if required.

Code

```
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 style of configuration, the equivalent to that example would be:

Code

```
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):

Code

```
from picamera2 import Picamera2

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

We discuss setting control values at runtime in Section 5.1. How to set camera controls.

4.3. Configuring a USB Camera

Picamera2 has limited supported for USB cameras such as webcams. You can connect several USB cameras and CSI2 cameras (the latter to a Pi's dedicated camera ports) at the same time, so long as the necessary ports and connectors are available. For more information, please refer to Section 8.7. Multiple Cameras.

You can create the `Picamera2` object in the usual way, but only the *main* stream will be available. The supported formats will depend on the camera, but *Picamera2* can in principle deal with both MJPEG and YUYV cameras, and where the camera supports both you can select by requesting the format "MJPEG" or "YUYV".

USB cameras can only use the software-rendered Qt preview window (`Preview.QT`). None of the hardware assisted rendering is supported. MJPEG streams can be rendered directly, but YUYV would require *OpenCV* to be installed in order to convert the image into a format that *Qt* understands. Both cases will use a significant extra amount of CPU.

The `capture_buffer` method will give you the raw camera data for each frame (a JPEG bitstream from an MJPEG camera, or an uncompressed YUYV image from a YUYV camera). A simple example:

Code

```
from picamera2 import Picamera2, Preview
```

```
picam2 = Picamera2()
config = picam2.create_preview_configuration({"format": "MJPEG"})
picam2.configure(config)

picam2.start_preview(Preview.QT)
picam2.start()

jpeg_buffer = picam2.capture_buffer()
```

If you have multiple cameras and need to discover which camera to open, please use the `Picamera2.global_camera_info` method.

In general, users should assume that other features, such as video recording, camera controls that are supported on Raspberry Pi cameras, and so forth, are not available. Hot-plugging of USB cameras is also not supported - Picamera2 should be completely shut down and restarted when cameras are added or removed.

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 passed the transform in to the `create_preview_configuration` function with `transform=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.

5. Camera controls and properties

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 C. Camera controls, and also by inspecting the `camera_controls` property of the `Picamera2` object:

```
Code
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 be overwritten by the camera tuning as soon as 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 associated 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. 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. Setting controls as part of the configuration

We have seen an example of this in Section 4.2.3. Configurations and runtime camera controls, 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.

5.1.2. Setting controls before the camera starts

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

```
Code
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.

5.1.3. 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.

```
Code
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, so would have to check the frame's metadata.

5.2. Object syntax for camera controls

We saw previously how control values can be associated with a particular camera configuration. When using the embedded `preview_configuration`, `video_configuration` and `still_configuration` objects we can achieve the same effect by setting the controls that are contained within each of those configuration objects. These controls can be used for exactly the same reason - to further customise the camera behaviour each time that configuration is applied. Once applied, there is no reason to further update them unless the application has chosen to define a new camera configuration.

When you want to set controls after creating the configuration, there is a separate embedded instance of the `Controls` class inside the `Picamera2` object that allows controls to be set either before starting the camera, or once it is running. For example, to set controls after configuration but before starting the camera:

```
Code
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:

```
Code
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.

5.3. Autofocus Controls

Autofocus controls obey the same general rules as all other controls, however, some guidance will be necessary before they can be used effectively. These controls should work correctly so long as the version of libcamera being used (such as that supplied by Raspberry Pi) implements *libcamera*'s published autofocus API correctly, and the attached camera module actually has autofocus (such as the *Raspberry Pi Camera Module 3*).

Camera modules that do not support autofocus (including earlier Raspberry Pi camera modules and the HQ camera) will not advertise these options as being available (in the `Picamera2.camera_controls` property), and attempting to set them will fail.

5.3.1. Autofocus Modes and State

The autofocus (AF) state machine has 3 *modes*, and its activity in each of these modes can be monitored by reading the `"AfState"` metadata that is returned with each image. The 3 modes are:

- `Manual` - The lens will never move spontaneously, but the `"LensPosition"` control can be used to move the lens "manually". The units for this control are dioptres (1 / *distance in metres*), so that zero can be used to denote "infinity". The `"LensPosition"` can be monitored in the image metadata too, and will indicate when the lens has reached the requested location.
- `Auto` - In this mode the `"AfTrigger"` control can be used to start an autofocus cycle. The `"AfState"` metadata that is received with images can be inspected to determine when this finishes and whether it was successful, though we recommend the use of helper functions that save the user from having to implement this. In this mode too, the lens will never move spontaneously until it is "triggered" by the application.
- `Continuous` - The autofocus algorithm will run continuously, and refocus spontaneously when necessary.

Applications are free to switch between these modes as required.

5.3.2. Continuous Autofocus

For example, to put the camera into continuous autofocus mode:

Code

```
from picamera2 import Picamera2
from libcamera import controls

picam2 = Picamera2()
picam2.start(show_preview=True)
picam2.set_controls({"AfMode": controls.AfModeEnum.Continuous})
```

5.3.3. Setting the Lens Position Manually

To put the camera into manual mode and set the lens position to infinity:

Code

```
from picamera2 import Picamera2
from libcamera import controls

picam2 = Picamera2()
picam2.start(show_preview=True)
picam2.set_controls({"AfMode": controls.AfModeEnum.Manual, "LensPosition": 0.0})
```

The lens position control (use `picam2.camera_controls['LensPosition']`) gives three values which are the minimum, maximum and default lens positions. The minimum value defines the furthest focal distance, and the maximum specifies the closest achieveable focal distance (by taking its reciprocal). The third value gives a "default" value, which is normally the hyperfocal position of the lens.

The minimum value for the lens position is most commonly 0.0 (meaning infinity). For the maximum, a value of 10.0 would indicate that the closest focal distance is 1 / 10 metres, or 10cm. Default values might often be around 0.5 to 1.0, implying a hyperfocal distance of approximately 1 to 2m.

In general, users should expect the distance calibrations to be approximate as it will depend on the accuracy of the tuning and the degree of variation between the user's module and the module for which the calibration was performed.

5.3.4. Triggering an Autofocus Cycle

For triggering an autofocus cycle in `Auto` mode, we recommend using a helper function that monitors the autofocus algorithm state for you, handles any complexities in the state transitions, and returns when the AF cycle is complete:

```
Code
from picamera2 import Picamera2
from libcamera import controls

picam2 = Picamera2()
picam2.start(show_preview=True)
success = picam2.autofocus_cycle()
```

The function returns `True` if the lens focused successfully, otherwise `False`. Should an application wish to avoid blocking while the autofocus cycle runs, we recommend replacing the final line (`success = picam2.autofocus_cycle()`) by

```
Code
job = picam2.autofocus_cycle(wait=False)

# Now do some other things, and when you finally want to be sure the autofocus
# cycle is finished:
success = picam2.wait(job)
```

This is in fact the normal method for running requests asynchronously - please see see the Section 6.5. Asynchronous capture on asynchronous capture for more details.

5.3.5. Other Autofocus Controls

The other standard *libcamera* autofocus controls are also supported, including:

- `"AfRange"` - adjust the focal distances that the algorithm searches
- `"AfSpeed"` - try to run the autofocus algorithm faster or slower
- `"AfMetering"` and `"AfWindows"` - lets the user change the area of the image used for focus.

To find out more about these controls, please consult Appendix C. Camera controls or the *libcamera* documentation and search for `Af`.

Finally, there is also a git_url_macro:picamera2[file=apps/app_capture_af.py, alt_text=*Qt* application] that demonstrates the use of the autofocus API.

5.4. 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:

```
Code
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 D. Camera properties.

5.5. 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.

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 seamlessly.

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

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

```
Code
from picamera2 import Picamera2
import time

picam2 = Picamera2()
picam2.start()

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

Although we regard this as a two-dimensional image, numpy will often report a third dimension of size three or four depending on whether every pixel is represented by three channels (RGB) or four 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
- `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.

Code

```
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 can also 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.

Code

```
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:

Code

```
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)
```

Note

Picamera2 versions from 0.3.26 onwards can encode YUV420 images directly to JPEG. Doing so is both faster and uses considerably less memory, making it particularly beneficial for lower spec Raspberry Pis. Therefore, if you do not need the captured image as an RGB image array, and are saving as JPEG, we recommend using something like this:

Code

```
capture_config = picam2.create_still_configuration({'format': 'YUV420'})
```

Temporal Denoise and Pi 5

From Pi 5 onwards, temporal denoise is supported. Therefore it can sometimes be beneficial to allow several frames to go by after a mode switch so that temporal denoise can start to operate. This is enabled by the `switch_mode_and_capture` family of methods with a `delay` parameter, which indicates how many frames to skip before actually capturing (and switching back).

In the earlier example, we would simply replace

```
Code
array = picam2.switch_mode_and_capture_array(capture_config, "main")
```

by (for example)

```
Code
array = picam2.switch_mode_and_capture_array(capture_config, "main", delay=10)
```

This is entirely optional in most circumstances, though we do recommend it for Pi 5 HDR captures.

On Pi 4 or earlier devices, there is no benefit in setting the delay parameter.

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:

```
Code
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:

```
Code
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'`.

Setting the Image Quality

The image quality can be set globally in the *Picamera2* object's `options` field (though can be changed while *Picamera2* is running), as listed in the table below.

Option name	Default value	Description
`quality`	90	JPEG quality level, where 0 is the worst quality and 95 is best.
`compress_level`	1	PNG compression level, where 0 gives no compression, 1 is the fastest that actually does any compression, and 9 is the slowest.

Please refer to the Python PIL module for more information.

For example, you can change these default quality parameters as follows:

```
Code
from picamera2 import Picamera2
import time

picam2 = Picamera2()
picam2.options["quality"] = 95
picam2.options["compress_level"] = 2
picam2.start()

time.sleep(1)
picam2.capture_file("test.jpg")
picam2.capture_file("test.png")
```

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:

```
Code
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:

```
Code
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.

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:

```
Code
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 convenient 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:

```
Code
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 facilitate 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:

```
Code
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:

```
Code
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 correspondence 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`

From Picamera2 version 0.3.19 onwards, you can capture requests with a context manager using the `captured_request()` method. This releases the request back to the camera system automatically, using the following pattern.

```
Code
with picam2.captured_request() as request:
    # Do something with the request.
    print(request)
# The request is released automatically when we leave the scope of the "with".
```

6.4.1. Capturing Requests at Specific Times

Sometimes there is a need to capture an image *after* some event has happened, perhaps a light being turned on or off. In these cases it's important to know that no part of the image started being exposed before the event in question happened. How can we do this?

When we capture a request we can access its metadata, which gives us a `SensorTimestamp`. This is the time, measured in nanoseconds from when the system booted, that the first pixel was read out of the sensor. So we have to *further* subtract the image exposure time to find out when the first pixel started being exposed.

The `Picamera2.capture_request` method makes this easy for us. By setting the `flush` parameter to `True` we can invoke exactly this behaviour - that the first pixel started being exposed no earlier than the moment we call the function. Alternatively, we can pass an explicit timestamp in nanoseconds if we have a slightly different instant in time in mind.

So for example

```
Code
request = picam2.capture_request(flush=True)
```

is equivalent to

```
Code
request = picam2.capture_request(flush=time.monotonic_ns())
```

6.4.2. 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 less 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.

6.4.3. Capturing Synchronised Still Images with Multiple Cameras

Raspberry Pi's *libcamera* implementation contains the ability to synchronise multiple cameras together using software. To summarise how this feature works:

- One camera is a *server*, and broadcasts timing information onto the network. (The network address is specified in the camera tuning file; all synchronised devices must share the same value. You can have multiple servers on a network, so long as they are using different addresses.)
- Other cameras are *clients* and listen out for these packets, whereupon they adjust their framerates so that, as far as possible, their frame start times match those of the server. They should all be configured to run at the same nominal fixed framerate (such as 30fps).
- This fixed framerate should be significantly *less* than the maximum framerate that the clients can achieve. This is so that they can be "sped up" from time to time to catch up with the server.
- Often there is just one client, but there can be an arbitrary number.
- Often a client will be another camera attached to the same Pi as the server, though they can also be on different devices. In such cases, the accuracy of the synchronisation will depend on the closeness of the their respective clocks. Clocks would normally be synchronised independently of the camera system, for example using NTP or PTP.
- The camera models do not need to be the same, though different cameras may not be able to run at exactly the same framerate, so synchronisation may not be as good.
- All devices, server and clients, are notified through image metadata (the `"SyncReady"` item) when the mutually agreed moment when everything is synchronised, occurs.
- To make a device act as the server, set the `SyncMode` control to `libcamera.controls.rpi.SyncModeEnum.Server`.
  For clients, use `libcamera.controls.rpi.SyncModeEnum.Client` instead.
- On the server, you can adjust the number of frames the server will run before everything is "synchronised" by setting the `"SyncFrames"` control. Note that there is no back-channel from clients to the server, so clients must be started "in good time" (and we would normally recommend starting clients first as they will just wait for the server to appear).
- Cameras should be configured with a `buffer_count` high enough to avoid dropping frames. In practice this will mean at least 3, but possibly more. Synchronisation still largely works when frames are being dropped, but obviously it will become difficult to match client and server frames up when some are missing (even though the frame timestamps will help here).

So to capture synchronised still images, we must start both a client and a server, and then wait until they indicate that they are synchronised. We can do this conveniently using `capture_sync_request()`, which waits for the first camera image where devices are synchronised, and returns it to the caller.

```
Code
from libcamera import controls
```

```
from picamera2 import Picamera2

picam2 = Picamera2()
ctrls = {'FrameRate': 30.0, 'SyncMode': controls.rpi.SyncModeEnum.Server}
# preview configurations get 4 buffers by default
config = picam2.create_preview_configuration(controls=ctrls)

picam2.start(config)
req = picam2.capture_sync_request()
print("Sync error:", req.get_metadata()['SyncTimer'])
```

On the clients, you will need to replace `controls.rpi.SyncModeEnum.Server` by `controls.rpi.SyncModeEnum.Client` but otherwise the script is identical. Note the `"SyncTimer"` metadata, which (when present) informs applications how long there is to go (in microseconds) until the agreed sychronisation point. On the frame captured at that precise moment, therefore, it provides an estimate of the error between the frame start times of client and server (as shown in the example).

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
- `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).

When you call a function in the usual blocking manner, the function in question will obviously return its "normal" result. When called in a non-blocking manner, the function will return a handle to a *job* which is what you will need if you want to block later on for the *job* to complete.

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, you should have recorded the *job* handle that was returned to you. An application may then call `Picamera2.wait(job)` to complete the process, for example:

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

The `wait` function returns the result that would have been returned if the operation had blocked initially. You don't have to wait for one job to complete before submitting another. They will complete in the order they were submitted.

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:

```
Code
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)
job = 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(job)
```

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:

```
Code
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 overwrite 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`). If 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:

```
Code
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 ten files named `test0.jpg` through `test9.jpg`, with a five-second delay before every capture.

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

```
Code
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 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.

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 to minimise the risk of dropping camera frames.

Here is a first example of capturing a ten-second video.

```
Code
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 interpreted 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:

```
Code
picam2.start_encoder(encoder, output)
picam2.start()
```

and `stop_recording` by:

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

7.1. Encoders

Encode Quality

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:

```
Code
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=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 below.

Note

On Pi 4 and earlier devices there is dedicated hardware for H264 and MJPEG encoding. On a Pi 5, however, these codecs are implemented in software using FFmpeg libraries. The performance on a Pi 5 is similar or better, and the images are encoded with better quality. The JPEG encoder runs in software on all platforms.

Encode Frame Rate

Normally, the encoder of necessity runs at the same frame rate as the camera. By default, every received camera frame gets sent to the encoder. However, you can use the encoder `frame_skip_count` property to instead receive every n^th^ frame.

For example, add the following code after creating the encoder in the example above to run the encode at half the rate of the camera:

```
Code
encoder.frame_skip_count = 2
```

Audio

Audio can be encoded and added to a video recording either by using the `FfmpegOutput` class or by using the `PyavOutput` class.

When using the `FfmpegOutput`, the audio is controlled by the `FfmpegOutput` object itself, so for more information please refer to Section 7.2.2. FfmpegOutput.

With the `PyavOutput`, audio is encoded by the encoder object. Please refer to Section 7.2.4. PyavOutput for more information on how to configure the encoder in this case.

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 Intra frame (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-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"`).

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 `None`) - the software will select the correct "colour space" for the stream being encoded so this parameter should normally be left blank.
- `colour_subsampling` (default `420`) - this is the form of YUV that the encoder will convert the RGB pixels into internally before encoding. It therefore determines whether a JPEG decoder will see a YUV420 image or something else. Valid values are `444` (YUV444), `422` (YUV422), `440` (YUV440), `420` (YUV420, the default), `411` (YUV411) or `Gray` (greyscale).

This encoder can accept either three-channel RGB (`"RGB888"` or `"BGR888"`) or 4-channel RGBA (`"XBGR8888"` or `"XRGB8888"`). Note that you cannot pass it any YUV formats.

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 Raspberry 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"`).

7.1.4. "Null" Encoder

The base `Encoder` class can be used as the "null" encoder, that is, an encoder that does nothing at all. It outputs exactly the same frames as were passed to it, without any compression or processing whatsoever. It could be used to record YUV or RGB frames (according to the output format of the stream being recorded), or even, as in the following example, the raw Bayer frames that are being output by the image sensor.

Code

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

picam2 = Picamera2()
config = picam2.create_video_configuration(raw={}, encode="raw")
picam2.configure(config)

encoder = Encoder()
picam2.start_recording(encoder, "test.raw")
time.sleep(5)
picam2.stop_recording()
```

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:

```
Code
from picamera2.outputs import FileOutput

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

A memory buffer:

```
Code
from picamera2.outputs import FileOutput
import io

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

A UDP network socket:

```
Code
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 substantial 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 can be documented 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:

```
Code
from picamera2.outputs import FfmpegOutput

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

Note

One limitation of the `FfmpegOutput` class is that there is no 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 frame rate. So 150 buffers is enough for five seconds 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:

```
Code
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()
```

7.2.4. PyavOutput

The `PyavOutput` is a more recent integration of Picamera2 with FFmpeg. Rather than passing frames to an external FFmpeg process, we pass them directly to the FFmpeg libraries running in the same Python process using the PyAV Python bindings.

Like the `FfmpegOutput`, the `PyavOutput` has a wealth of possible options which would require detailed knowledge of the PyAV module, and we shall give just a few simple examples here. However, further documentation is available at the PyAV documentation page and GitHub repository.

The `PyavOutput` constructor has a single required arugment, the `output_filename`, and the rest are optional:

- `output_filename` - the name of the output file or destination, typically a regular filename (such as `"test.mp4"`), or maybe a socket (for example `"udp://<ip-addr>:<port>"`).
- `format` - a string identifying the FFmpeg output format. This is often deduced from the `output_filename` (for example, if it ends with `".mp4"`), but otherwise it can be specified here. For an `mp4` file, use `format="mp4"`, or for an MPEG2 transport stream, use `format="mpegts"`. For an unformatted H.264 bitstream, you can use `format="h264"`.

The `PyavOutput` supports audio too (if the chosen output format does). However, because there is no external FFmpeg process producing the audio, we have to get the encoder to create and supply audio packets. The base `Encoder` class therefore has the following properties.

- `audio` - default `False`. Set to `True` for the encoder to supply encoded audio packets to the `PyavOutput`.
- `audio_input` - default `{'file': 'default', 'format': 'pulse'}`. These parameters are passed to PyAV to create its audio input container. Any parameters that are understood by `av.open` can be used here.
- `audio_output` - default `{'codec_name': 'aac'}`. ThesAV to create the audio output stream. Any parameters that are understood by `av.container.Container.add_stream` can be used here.
- `audio_sync` - default `-100000`. The amount by which to delay the audio stream so that it is synchronised with the video. This number is in units of microseconds.

Note that the `PyavOutput` (and the associated `CircularOutput2` when connected to a `PyavOutput`, which we shall learn about in a moment) are the *only* output types that can accept audio packets from the encoder.

Examples

To record a short mp4 file with audio:

Code

```
import time

from picamera2 import Picamera2
from picamera2.encoders import H264Encoder
from picamera2.outputs import PyavOutput

picam2 = Picamera2()
config = picam2.create_video_configuration({'size': (1280, 720), 'format': 'YUV420'})
picam2.configure(config)

encoder = H264Encoder(bitrate=10000000)
encoder.audio = True
output = PyavOutput("test.mp4")
picam2.start_recording(encoder, output)

time.sleep(5)

picam2.stop_recording()
```

The `PyavOutput` has an `error_callback` which is invoked when any kind of error occurs. The following script will wait for TCP connections and send an MPEG2 transport stream until the client disconnects.

Code

```
import socket
from threading import Event

from picamera2 import Picamera2
from picamera2.encoders import H264Encoder
from picamera2.outputs import PyavOutput

picam2 = Picamera2()
video_config = picam2.create_video_configuration({"size": (1280, 720), 'format': 'YUV420'})
picam2.configure(video_config)

encoder = H264Encoder(bitrate=10000000)
encoder.audio = True

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(("0.0.0.0", 8888))
```

```
    while True:
        sock.listen()
        conn, addr = sock.accept()  # client connected

        output = PyavOutput(f"pipe:{conn.fileno()}", format="mpegts")
        event = Event()
        output.error_callback = lambda e: event.set()

        picam2.start_recording(encoder, output)

        event.wait()  # wait for client to disconnect

        picam2.stop_recording()
```

7.2.5. CircularOutput2

The original `CircularOutput` class can only write to `FileOutput` types of output. Therefore we need a revised version of the class to work with the `PyavOutput`. The `CircularOutput2` can actually work with any type of output, but it's in conjunction with the `PyavOutput` that it's most useful.

The `CircularOutput2` constructor allows you to specify `buffer_duration_ms` - effectively the size of the available buffer, but specified as a length of time. The `buffer_duration_ms` property can even be updated while the circular output is running. It has the following two methods availalbe to applications:

- `open_output(output)` - pass another output object to which the circular buffer will start writing (with the delay given by `buffer_duration_ms`, of course). This is equivalent to setting the `fileoutput` property of the old `CircularOutput`.
- `close_output()` - to stop writing to the previously opened output, and close it. If an open output is *not* closed before the `CircularOutput2` itself is stopped, then the remaining contents of the buffer (up to `buffer_duration_ms` worth of time) will be flushed to that output after which it will be closed automatically too.

Here's a short example showing how we can record time-shifted video and audio together to mp4 files.

Code

```
import time

from picamera2 import Picamera2
from picamera2.encoders import H264Encoder
from picamera2.outputs import CircularOutput2, PyavOutput

picam2 = Picamera2()
config = picam2.create_video_configuration({'size': (1280, 720), 'format': 'YUV420'})
picam2.configure(config)

encoder = H264Encoder(bitrate=10000000)
encoder.audio = True
circular = CircularOutput2(buffer_duration_ms=5000)
picam2.start_recording(encoder, circular)

time.sleep(5)

circular.open_output(PyavOutput("start.mp4"))
time.sleep(5)
circular.close_output()

circular.open_output(PyavOutput("end.mp4"))
picam2.stop_recording()
```

In this example, the recording of "start.mp4" begins 5 seconds after we start recording to the circular buffer, but because of the 5 seconds length of this buffer, it will actually capture the first 5 seconds of the video.

"end.mp4" is not closed before the circular output is stopped, so all the remaining contents of the circular buffer (the last 5 seconds in this case) will be flushed out to it.

7.2.6. SplittableOuptut

Imagine that you want to stop recording an output file that you're currently writing to, and start recording to a new file instead. You can do this easily enough by stopping the current output and opening a new one, but you will find that some of the camera frames are going to get dropped while the switchover happens.

The `SplittableOutput` gives you a way of switching seamlessly from one output file to another. It accomplishes this by getting the new output ready, and then waiting for a video key frame ("I frame"), which is the point at which we can re-direct to the new output without losing anything. Once the new output has started receiving frames, it's safe to finish and close the previous output.

Here's an example showing how to switch from one mp4 file to another.

Code

```
import time

from picamera2 import Picamera2
from picamera2.encoders import H264Encoder
from picamera2.outputs import PyavOutput, SplittableOutput

picam2 = Picamera2()
config = picam2.create_video_configuration()
picam2.configure(config)
encoder = H264Encoder(bitrate=5000000)

splitter = SplittableOutput(output=PyavOutput("test1.mp4"))

picam2.start_recording(encoder, splitter)

time.sleep(5)

print("Waiting for switchover...")
splitter.split_output(PyavOutput("test2.mp4"))
print("Switched to new output!")

time.sleep(5)

picam2.stop_recording()
```

In this example:

- We start writing initially to the file `test1.mp4` (when we create the `SplittableOutput`). Had we used just `splitter = SplittableOutput()` then nothing would be recorded until we reach the `split_output` call later.
- The `split_output` call only returns once the new output is being written to, and the previous output (if any) has been closed. So it will block for a length of time, depending mostly on the time until the next video key frame.
- We can use `split_output` as many times as we want to switch outputs, until the recording is stopped.

7.3. Capturing Synchronised Videos with Multiple Cameras

As discussed previously, *libcamera* is able to synchronise multiple cameras through software. Readers are referred to Section 6.4.3. Capturing Synchronised Still Images with Multiple Cameras about still image capture for a reminder how this works.

The same underlying features allow video recordings to be synchronised frame by frame. The `Encoder` class will monitor the image metadata and start recording only once the synchronisation signal is received. All we have to do is set the `sync_enable` flag. The encoder also provides a "sync" `Event` which we can check to know when the synchronised recording has actually started.

The example below would record a 5 second synchronised video between a server and a number of clients.

Code

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

picam2 = Picamera2()
```

```
ctrls = {'SyncMode': controls.rpi.SyncModeEnum.Server, 'FrameRate': 30}
config = picam2.create_video_configuration(controls=ctrls)
picam2.configure(config)
encoder = H264Encoder(bitrate=5000000)
encoder.sync_enable = True  # enable synchronised recording
output = "server.h264"

picam2.start_recording(encoder, output)
print("Waiting for sync...")
encoder.sync.wait()
print("Recording has started")
time.sleep(5)
picam2.stop_recording()
print("Recording has finished")
```

For the clients, replace `controls.rpi.SyncModeEnum.Server` by `controls.rpi.SyncModeEnum.Client` (and probably the name of the output file "server.h264"), but the script is otherwise the same.

7.4. 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 recording duration. 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 when a microphone is attached as the default PulseAudio input.

This example will record a five-second MP4 file:

```
Code
from picamera2 import Picamera2

picam2 = Picamera2()

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

7.5. 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.

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 three-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.

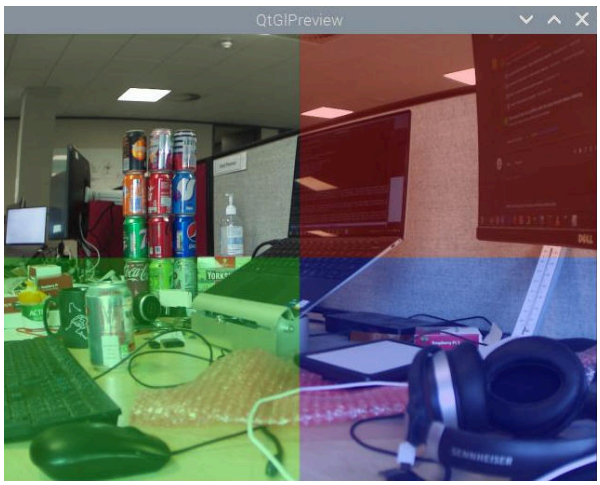
Here is a very simple example:

Code

```
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. Remember that, if loading an RGBA image with OpenCV, you need to use the `IMREAD_UNCHANGED` flag:

Code

```
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.

Code

```
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 and can cause a deadlock.

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 on 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:

. Every time a completed request is received from the camera, it calls the first function in the list. . 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.) . 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:

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))]
    return picam2.dispatch_functions(functions, wait=True)

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.

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 choose, the size of a single full resolution 12MP image, and whether they work with certain other modules.

	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	Yes
PNG 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 4 and earlier models it provides memory that can be used directly by the camera system and all its hardware devices. For Pi 5 (and later models) CMA memory is not used, and this discussion can be ignored.

For Pis that use CMA for the camera, 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 with:

- `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 provided 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.

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:

```
Code
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_YUV420p2RGB)
```

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 four buffers by default. This is normally enough to keep the camera system running smoothly even when there is moderate additional processing going on.
- Still capture configurations are given only one 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 six 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).

This behaviour (where *Picamera2* holds on to the last camera frame) is customisable in the camera configuration via the `queue` parameter. If you want to guarantee that every capture request returned to you arrived from the camera system *after* the request was made, then this parameter should be passed as `False`. Please refer to the queue parameter documentation for more details.

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 affect 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:

```
Code
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:

. Functions that return immediately and are safe to call . 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, and . Functions that should not be called at all.

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

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_sync_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
`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.
- The *job* that was started by the non-blocking call will be passed to the function connected to the `done_signal`. The application can call `Picamera2.wait` with this *job* to obtain the return value of the original call.

We finish with a small worked example of this:

Code

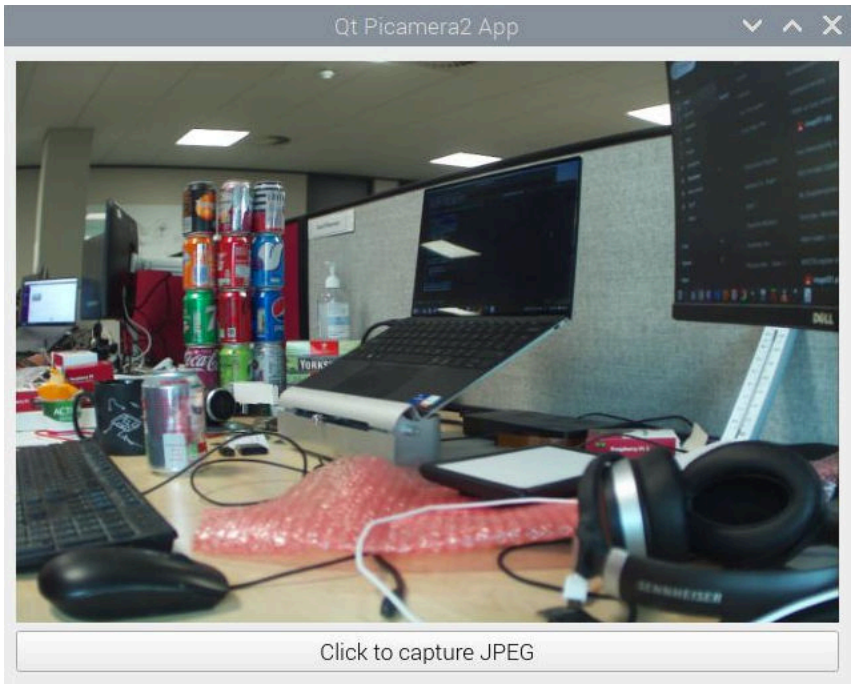
```
from PyQt5 import QtWidgets
from PyQt5.QtWidgets import QPushButton, QVBoxLayout, QApplication, QWidget

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(job):
    result = picam2.wait(job)
    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 can call the `wait(job)` method if there is a result that we need.

And here is our sophisticated Qt application in action:

Figure 4.
A Qt app using Picamera2



Support for PyQt6, PySide2 and PySide6

The Picamera2 widgets are by default implemented using *PyQt5*. However, if you find yourself writing a GUI application using a different version of the *Qt* toolkit, further versions of the Picamera2 widgets are available.

For example, if you need to use *PyQt6*, replace the line

```
Code
from picamera2.previews.qt import QGlPicamera2
```

by

```
Code
from picamera2.previews.qt import QGl6Picamera2 as QGlPicamera2
```

and carry on using `QGlPicamera2` in your code as before.

To use *PySide6* instead, replace the original line by

```
Code
from picamera2.previews.qt import QGlSide6Picamera2 as QGlPicamera2
```

and for *PySide2*

```
Code
from picamera2.previews.qt import QGlSide2Picamera2 as QGlPicamera2
```

The same substitutions also hold for the `QPicamera2` widget, for example `Q6Picamera2` or `QSide6Picamera2`.

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.

8.6. Debug Logging

8.6.1. *Picamera2* Debug

Picamera2 uses the Python *logging* module. It creates a *logger* under the name `picamera2` but assigns no debug level or handlers to it as this is generally regarded as being within the remit of application code.

Users familiar with the *logging* module can assign handlers and set the level in the usual way. Users not familiar with the *logging* module, and who simply want to see some debug statements being printed out, can use the `Picamera2.set_logging` function, for example:

```
Code
from picamera2 import Picamera2

Picamera2.set_logging(Picamera2.DEBUG)
```

which would set the logging level to output all `DEBUG` (or more serious) messages. (`Picamera2.DEBUG` is a synonym for `logging.DEBUG` but saves you the extra `import logging`.)

Besides the logging level, the `set_logging` function also allows you to specify:

- `output` - the destination for the messages, which defaults to `sys.stderr`.
- `msg` - the *logging* module format string for the message, defaulting to `"%(name)s %(levelname)s: %(message)s"`.

For more information on these parameters, please consult the *logging* module documentation.

Note

Older versions of *Picamera2* had a `verbose_console` parameter in the `Picamera2` class constructor, and which would set up the logging level. This has been deprecated and no longer does anything. The `Picamera2.set_logging` function should be used instead.

8.6.2. *libcamera* Debug

libcamera, the C++ library underpinning *Picamera2*, has its own logging system. Although formally beyond the scope of the *Picamera2* documentation we give some brief notes on how to use it.

The *libcamera* logging system is controlled by two main environment variables:

- `LIBCAMERA_LOG_FILE` - the name of a file to which to send the log output. If unspecified, logging messages will go to `stderr`.
- `LIBCAMERA_LOG_LEVELS` - specifies which parts of *libcamera* should output logging, and at what level.

Although it has more sophisticated usages, `LIBCAMERA_LOG_LEVELS` can simply be set to one of a range of fixed values (or *levels*) which then apply to all the modules within *libcamera*. These levels may be given as numbers or as (easier to remember) strings. For each logging level, messages at that level *or any more serious levels* will be emitted. These levels are:

- `DEBUG` or `0` - output lots of debug and all other messages
- `INFO` or `1` - output informational or more serious messages
- `WARN` or `2` - output warning or more seious (error) messages
- `ERROR` or `3` - output error or fatal error messages
- `FATAL` or `4` - output only fatal error messages.

For example, to run *Picamera2* with only warning or error messages, you might start Python like this:

```
Code
LIBCAMERA_LOG_LEVELS=WARN python
```

You can of course set `LIBCAMERA_LOG_LEVELS` permanently in your user profile (for example, your `.bashrc` file).

8.7. Multiple Cameras

If you have a compute module such as a CM4, and appropriate I/O connections, you can attach two cameras to the Raspberry Pi and drive them simultaneously from within a single Python session. Even on regular Pis, there is some support for USB cameras which can be managed by *Picamera2* in the same way.

The process is essentially identical to using a single camera, except that you create multiple `Picamera2` objects and then configure and start them independently.

The `Picamera2.global_camera_info()` Method

Before creating the `Picamera2` objects, you can call the `Picamera2.global_camera_info()` method to find out what cameras are attached. This returns a list containing one dictionary for each camera, ordered according the camera number you would pass to the `Picamera2` constructor to open that device. The dictionary contains:

- "Model" - the model name of the camera, as advertised by the camera driver.
- "Location" - a number reporting how the camera is mounted, as reported by *libcamera*.
- "Rotation" - how the camera is rotated for normal operation, as reported by *libcamera*.
- "Id" - an identifier string for the camera, indicating how the camera is connected. You can tell from this value whether the camera is accessed using I2C or USB.

You should always check this list to discover which camera is which as the order can change when the system boots or USB cameras are re-connected. However, you can rely on CSI2 cameras (attached to the dedicated camera port) coming ahead of any USB cameras, as they are probed earlier in the boot process.

The following example would start both cameras and capture a JPEG image from each.

Code

```
from picamera2 import Picamera2

picam2a = Picamera2(0)
picam2b = Picamera2(1)

picam2a.start()
picam2b.start()

picam2a.capture_file("cam0.jpg")
picam2b.capture_file("cam1.jpg")

picam2a.stop()
picam2b.stop()
```

Because the cameras run independently, there is no form of synchronisation of any kind between them and they may be of completely different types (for example a Raspberry Pi v2 camera and an HQ camera, or a Raspberry Pi camera and a USB webcam).

You can also use bridging devices to connect multiple cameras to a single Raspberry Pi camera port as long as you have appropriate `dtoverlay` files from the supplier. However, you cannot drive them simultaneously (you can only open one camera at a time).

8.8. Multi-Processing

Multi-Threading

There are no restrictions in Picamera2 on passing image buffers to other threads in your application for processing. In general, you shouldn't try to *control* the camera (that is, start, stop or re-configure it, which includes mode-switching) from multiple threads unless you are taking care to lock other threads out while one thread is actively performing one of these operations.

Multi-Processing

But we know that multi-threaded applications don't always show the performance improvements that we expect because of Python's GIL (Global Interpreter Lock). In essence, this means only one Python thread can run at once. One way to circumvent this is to use Python *multi-processing* which *does* allow us to run code on multiple cores simultaneously, albeit in separate processes.

But before diving into the multi-processing options, it's worth considering whether it will actually help. In many cases, multi-threading works well because the heavy lifting within these other threads is performed by large (typically) C/C++ functions. These functions will normally *release* the GIL, allowing other Python threads (which may themselves call other large C/C++ functions)

to run. Typical examples would include OpenCV functions, or functions to evaluate the results of a passing an image to a neural network.

Remote Requests

Picamera2 provides a mechanism for passing *requests*, that is, objects of the class `CompletedRequest` and usually obtained by calling the `Picamera2.capture_request()` (or similar) method, to a child process. When the request is received in the child process, it is now an object of type `RemoteRequest`. This mirrors the `CompletedRequest` quite closely and permits many of the same operations - such as saving images, turning them into `numpy` arrays, or directly accessing pixels within the buffer (using `RemoteMappedArray` in place of the usual `MappedArray`).

The `RemoteRequest` also copies all the metadata that was available in the original request, and image buffers are transferred by mapping the underlying file descriptors, so these potentially very large amounts of data become "zero copy".

On the parent side, the `CompletedRequest` is held (and not returned to the camera system) until the child process has completely finished processing the corresponding `RemoteRequest`. This is because the child may be accessing the same underlying memory that is mapped by the parent.

Processes and Pools

While Picamera2 does implement a `Process` class which you can use to create a child process, it's often easier to create a `Pool` of child processes instead. Creating a `Pool` with just one process is equivalent to creating a single `Process` explicitly, so we'll concentrate on pools in the following short examples.

Code

```
from picamera2 import Picamera2, Pool

def run(request, image_num=None):
    request.save(f"image_{image_num}.jpg")

picam2 = Picamera2()
config = picam2.create_preview_configuration(buffer_count=4)
picam2.configure(config)

with Pool(run, 4, picam2) as pool:
    picam2.start(show_preview=True)

    for i in range(30):
        with picam2.captured_request() as request:
            pool.send(request, image_num=i)
```

Here we're saving 30 JPEG files, but delegating the job of encoding and saving to different processes. Observe:

- The `Pool` has to be given a function (`run` in this case) which each process will execute when a request is sent to it.
- We can call `save` on the `RemoteRequest`, just as we can on a `CompletedRequest`.
- The `run` function can accept extra parameters of the application's choice that are specified along with the `send` command (`image_num` in this case).
- Only when the `run` function returns will the parent recycle its corresponding request back to the camera system.
- We've created a pool with 4 child processes. Often it's good to match this to the number of requests (buffers) allocated to the camera system - adding more processes doesn't help if you don't have enough buffers to keep them all fed, and likewise having lots of buffers doesn't help if you're actually just waiting on remote processes all the time!
- We recommend creating the `Pool` before starting the camera (as above). Creating all the child processes is relatively expensive, and may cause camera timeouts if it's already running.

In this particular case we had no need to receive anything back from the child processes, so we'll look at that next.

Code

```
import queue
import threading

import numpy as np
from picamera2 import Picamera2, Pool, RemoteMappedArray

def run(request):
    with RemoteMappedArray(request, 'main') as m:
        average = np.average(m.array)
```

```
    return average

def return_thread_func(futures_queue):
    while (future := futures_queue.get()):
        result = future.result()
        print("Received", result)

picam2 = Picamera2()
config = picam2.create_preview_configuration(buffer_count=4)
picam2.configure(config)

futures = queue.Queue()
return_thread = threading.Thread(target=return_thread_func, args=(futures,))
return_thread.start()

with Pool(run, 4, picam2) as pool:
    picam2.start(show_preview=True)

    for _ in range(30):
        with picam2.captured_request() as request:
            future = pool.send(request)
            futures.put(future)

    futures.put(None)
    return_thread.join()
```

This follows the same pattern as before, with a `Pool` that is passed a `run` function. However, in this case:

- The `run` function returns a value which is sent back to the parent process.
- The `send` function actually returns a `Future`. The parent can wait on this for the result from the child process.
- We have used `RemoteMappedArray` (instead of `MappedArray`) to access the image buffer. This involves no copying of the pixel data.
- Of course, these results are returned asynchronously, so we have to decide how to handle them. In this case, we start another thread that waits on the futures and (in this example) simply outputs the results, though real applications are likely to do something more complex.
- Note that we need to stop the asynchronous results thread when we're done. "Joining" the thread guarantees that all the child processes are finished before exiting and closing the pool.

There are further examples in the Picamera2 repository. Please search for the file names that begin with `remote`.

9. Application notes

This section answers a selection of “how to” questions for particular use cases.

9.1. Streaming to a network

9.1.1. Simple Streaming

There are some simple streaming examples available:

- capture_stream.py - streaming to a TCP socket
- capture_stream_udp.py - streaming to a UDP socket
- pyav_stream2.py - streaming to a TCP socket using `PyavOutput`
- 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.

9.1.2. Streaming using MediaMTX

A good option for streaming is to use MediaMTX. MediaMTX is very capable so the webpage is perhaps a little daunting, but all you have to do is download the latest `linux_arm64v8` release (for recent models of Pi), unpack it, and the executable is ready to go. You don’t even need to use any Python, though we will later!

To stream the camera image, back-up the `mediamtx.yml` file (it contains documentation that will be useful later), and replace the contents by

```
Code
paths:
  cam:
    source: rpiCamera
    rpiCameraBitrate: 10000000
```

Here the stream bitrate is set to 10Mbps, but the original `mediamtx.yml` file documents many other Raspberry Pi camera options.

If want the camera stream to start only when requested, and stop when no one is viewing it any more, add the line

```
Code
    sourceOnDemand: yes
```

Clients can access the stream using a web browser by visiting `http://<ip-address>:8889/cam`. RTSP clients can request `rtsp://<ip-address>:8554/cam`.

Advanced Streaming with MediaMTX

The MediaMTX `rpiCamera` source is fine for streaming a plain camera image, but what if we want to do something else as well? For example, we might wish to annotate or change the camera image in some way, or perhaps perform other camera operations such as capturing a still image from time to time. Here we’re going to use some Python scripts.

First, replace the contents of the `mediamtx.yml` file by:

```
Code
paths:
  cam:
```

This tells MediaMTX that applications will ask for a stream named `cam`, as we saw previously. The `cam` stream will be taken from anything that “publishes” a stream named `cam` to MediaMTX. And we can publish a stream to MediaMTX by sending it an RTSP stream using the following script, which we start after launching MediaMTX.

```
Code
import time
```

```
from picamera2 import Picamera2
from picamera2.encoders import H264Encoder
from picamera2.outputs import PyavOutput

picam2 = Picamera2()
main = {'size': (1920, 1080), 'format': 'YUV420'}
controls = {'FrameRate': 30}
config = picam2.create_video_configuration(main, controls=controls)
picam2.configure(config)

encoder = H264Encoder(bitrate=10000000)
output = PyavOutput("rtsp://127.0.0.1:8554/cam", format="rtsp")

print("Camera starting")
picam2.start_recording(encoder, output)

try:
    while True:
        time.sleep(0.5)
except KeyboardInterrupt:
    print("Camera stopping")
```

MediaMTX will receive this stream, and then remux and forward it to any clients requesting it either via WebRTC (browsers) or RTSP.

There are a couple of things to notice about the RTSP address `rtsp://127.0.0.1:8554/cam` that we used. Firstly, `127.0.0.1` implies that MediaMTX is running on the same machine. This may be a sensible choice but we could run MediaMTX on a different machine altogether if we had a reason to do so. Secondly, the stream name `cam` at the end is what identifies this as the `cam` stream to MediaMTX. We could change this, for example, to `camera` if we wished, accompanied by corresponding changes to the `mediamtx.yml` file, and also to the URLs requested by clients.

Note

There can be packet loss streaming from the Python process to MediaMTX which causes pauses in the video. To fix this, we must increase the size of the network receive buffers by (as root) adding these lines

Code

```
net.core.rmem_max=1000000
net.core.rmem_default=1000000
```

to your `/etc/sysctl.conf` file (and rebooting).

We are then free to edit the Python script as we please to perform whatever other functions we have in mind. As before, if we again want the script only to start when a client requests the stream, we can add

Code

```
    runOnDemand: python /home/pi/stream.py
```

to the previous `mediamtx.yml` file (replacing `/home/pi/stream.py` by the name of your script).

Finally, the MediaMTX webpage gives detailed instructions on how to set it up as a daemon that starts automatically on boot, should you wish to do so.

9.2. Output using FFmpeg and PyAV

The `FfmpegOutput` class has been discussed previously, and more recently Picamera2 has also acquired the `PyavOutput` class which uses the Python `av` module. Here we give some more complex examples.

9.2.1. HLS live stream using FFmpeg

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

Code

```
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:

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

9.2.2. HLS live stream using PyAV

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

```
Code
from picamera2.outputs import PyavOutput

options = {
    "hls_time": "5",
    "hls_list_size": "8",
    "hls_flags": "delete_segments+append_list+independent_segments"
}
output = PyavOutput("/dev/shm/hls/stream.m3u8", format='hls', options=options)
```

You would also have to start an HTTP server to enable remote clients to access the stream as before. In this example, note the use of `/dev/shm` which writes to a memory-based file system, thereby saving wear on any attached physical media.

The use of the `PyavOutput` is generally recommended over the `FfmpegOutput`, as it doesn't involve piping frames to an external process that will have to re-timestamp the frames (and possibly introduce inaccuracies).

9.2.3. MPEG-DASH live stream

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

```
Code
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 an HTTP server just as we saw previously:

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

9.2.4. Sending an MPEG-2 transport stream to a socket

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

```
Code
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.

The newer `PyavOutput` class is often a better option here, because it interfaces directly to the FFmpeg libraries rather than invoking `ffmpeg` as a separate process. This means it will have more accurate timestamps. We can use it instead by replacing the above with

```
Code
from picamera2.outputs import PyavOutput

output = PyavOutput("udp://<ip-addr>:<port>", format="mpegts")
```

Note that `PyavOutput` supports audio too, but we have to tell the *encoder* to enable audio encoding. Enabling audio like this is only valid when the output is indeed a `PyavOutput` or the closely associated `CircularOutput2` class.

We finish with an example where we wait for a TCP socket connection, and then stream an MPEG2 transport stream to it. Audio is enabled.

Code

```
import socket
from threading import Event

from picamera2 import Picamera2
from picamera2.encoders import H264Encoder
from picamera2.outputs import PyavOutput

picam2 = Picamera2()
video_config = picam2.create_video_configuration({"size": (1280, 720), 'format': 'YUV420'})
picam2.configure(video_config)

encoder = H264Encoder(bitrate=10000000)
encoder.audio = True  # enable audio

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(("0.0.0.0", 8888))

    while True:
        sock.listen()  # wait for connection

        conn, addr = sock.accept()  # accept connection

        output = PyavOutput(f"pipe:{conn.fileno()}", format="mpegts")
        event = Event()
        output.error_callback = lambda e: event.set()  # notify of disconnection

        picam2.start_recording(encoder, output)

        event.wait()  # wait for disconnection

        picam2.stop_recording()
```

9.2.5. V4L2 Loopback Video

In conjunction with the `PyavOutput`, Picamera2 can also be used to create a V4L2 loopback video device.

First, create the loopback device itself with the correct parameters by entering the following in a terminal window.

Code

```
sudo modprobe v4l2loopback devices=1 video_nr=10 card_label="loopback" exclusive_caps=1 format="YUV420P"
```

which will create the new device as `/dev/video10`.

Now we can run a script to feed uncompressed video frames to the new device.

Code

```
import time
from picamera2 import Picamera2
from picamera2.encoders import Encoder
from picamera2.outputs import PyavOutput

picam2 = Picamera2()
main = {'format': 'YUV420', 'size': (1280, 720)}
controls = {'FrameRate': 30}
config = picam2.create_video_configuration(main, controls=controls)
picam2.configure(config)

encoder = Encoder(framerate=30)
```

```
output = PyavOutput("/dev/video10", format='v4l2')

picam2.start_recording(encoder, output)
while True:
    time.sleep(1)
```

You can now view the video stream with, for example, VLC. From the “Media” menu, choose “Open Capture Device…” and select `/dev/video10`.

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 five seconds we start the second output and record five seconds’ worth of H.264 video to a file. We close this output file, but the network stream continues to play.

Code

```
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 use to 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 on 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.

Code

```
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)
```

9.5. Using Python Virtual Environments

Python virtual environments should be created in such a way as to include your site packages. This is because the *libcamera* Python bindings are not available through *PyPI* so you can't install them explicitly afterwards. The correct command to do this would be:

Code

```
python -m venv --system-site-packages my-env
```

which would create a virtual environment named `my-env`.

9.6. HDR mode and the *Raspberry Pi Camera Module 3*

The *Raspberry Pi Camera Module 3* implements an HDR (High Dynamic Range) camera mode. Unfortunately V4L2 (the Linux video/camera kernel interface) does not readily support this kind of camera, where the sensor itself has HDR and non-HDR modes. For the moment, therefore, activating (or de-activating) the HDR mode has to be handled externally to the more usual *Picamera2* camera configuration methods.

These commands must be issued *before* opening the `Picamera2` object for the Camera Module 3. Note that the image sensor in the Camera Module 3 is a Sony IMX708.

Code

```
    from picamera2.devices.imx708 import IMX708

    cam = IMX708(camera_num)
    cam.set_sensor_hdr_mode(True)
    cam.close()
    # After this, we can do: picam2 = Picamera2(camera_num)
```

or

Code

```
    from picamera2.devices.imx708 import IMX708

    with IMX708(camera_num) as cam:
        cam.set_sensor_hdr_mode(True)
    # After this, we can do: picam2 = Picamera2(camera_num)
```

where `camera_num` is the id number for your Camera Module 3.

To disable HDR mode, use `False` instead of `True`. Any open `Picamera2` instance must be closed before repeating this process.

Alternatively, the HDR modes can be enabled or disabled *before* starting the Python interpreter. To enable it, use:

```
Code
v4l2-ctl --set-ctrl wide_dynamic_range=1 -d /dev/v4l-subdev0
```

or to disable it, use:

```
Code
v4l2-ctl --set-ctrl wide_dynamic_range=0 -d /dev/v4l-subdev0
```

Note

The sensor may not always be `v4l-subdev0` in which case you will have to discover the correct sub-device. On a Pi 5 in particular, the sensor is more likely to be `v4l-subdev2`, though again this can vary. To find the correct one, you can use (for example) `v4l2-ctl -d /dev/v4l-subdev0 -l` to list a device's controls, and select the one that has a `wide_dynamic_range` control.

9.7. HDR mode on the Pi 5

The Raspberry Pi 5 has some in-built support for HDR, allowing its use even for non-HDR cameras.

To turn on the HDR mode, use (assuming `picam2` is your Picamera2 object)

```
Code
import libcamera

picam2.set_controls({'HdrMode': libcamera.controls.HdrModeEnum.SingleExposure})
```

and to disable it use

```
Code
picam2.set_controls({'HdrMode': libcamera.controls.HdrModeEnum.Off})
```

The HDR mode can be configured in the {rpi-datasheets-url}/camera/raspberry-pi-camera-guide.pdf[Raspberry Pi Camera Tuning Guide] but which is beyond the scope of this document.

Because this HDR mode relies on the Pi 5's TDN (Temporal Denoise) function to accumulate images, it's important to skip a number of frames after a mode switch to allow this process to happen. When using the `switch_mode_and_capture` family of methods, the `delay` parameter should therefore be specified for this purpose, for example

```
Code
picam2.switch_mode_and_capture_file(capture_config, "image.jpg", delay=10)
```

would allow 10 frames to be accumulated.

9.8. Using the Hailo AI Accelerator

The Hailo accelerator should be installed according to the official instructions. Please follow them through to confirm that the supplied demos are working correctly.

The neural network model should be compiled into a HEF file. A number of HEF files are supplied with the Hailo AI Kit software, which can be found in `/usr/share/hailo-models`. The process for creating them lies beyond the scope of this guide, however, Hailo provide documentation explaining how models can optionally be re-trained and compiled into the HEF format.

Once you have a HEF file, it can be used as follows:

```
Code
from picamera2 import Picamera2
from picamera2.devices.hailo import Hailo

with Hailo("/path/to/hef-file") as hailo:
```

```
    model_h, model_w, _ = hailo.get_input_shape()

    picam2 = Picamera2()
    main = {'size': (model_w, model_h), 'format': 'RGB888'}
    config = picam2.create_preview_configuration(main)
    picam2.start(config)

    frame = picam2.capture_array()

    inference_results = hailo.run(frame)
```

Alternatively, the `Hailo` object can be created with `hailo = Hailo("/path/to/hef-file")`python and closed with `hailo.close()`.

In general, the application will need to know what the network does and how to interpret the results. Often, it may make sense to pass the low resolution image to the Hailo model, so that a larger “main” output can also be requested.

For more information, please refer to our example code.

9.9. Using the IMX500 AI Accelerator

The Sony IMX500 camera module should be connected in the usual way, and the software should be installed according to the official instructions. Please follow them through to confirm that the supplied demos are working correctly.

The neural network model should be packaged into a RPK file. A number of RPK files are supplied with the IMX500 software, which can be found in `/usr/share/imx500-models`. The process for creating them lies beyond the scope of this guide, however, Sony provide documentation and tutorials on how to do this on the ATRIOS developer website.

Once you have the RPK file, it can be used as follows:

```
Code
from picamera2 import Picamera2
from picamera2.devices import IMX500

imx500 = IMX500("/path/to/rpk-file")

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

metadata = picam2.capture_metadata()
network_outputs = imx500.get_outputs(metadata)
```

Note that the output tensor of the network is extracted from the camera frame metadata, and the application will in general have to know how to interpret and use the network results.

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)

Figure 5.
Different image formats.

	Bits per pixel	Optimal alignment	Shape	Description
XBGR8888	32	16	$(\text{height}, \text{width}, 4)$	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	$(\text{height}, \text{width}, 4)$	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	$(\text{height}, \text{width}, 3)$	RGB format. Each pixel is laid out as `[R, G, B]`.
RGB888	24	32	$(\text{height}, \text{width}, 3)$	RGB format. Each pixel is laid out as `[B, G, R]`.
YUV420	12	128 (Pi 5), 64 (non-Pi 5)	$(\frac{3}{2} \cdot \text{height}, \text{width})$	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 two rows of U (or V) values on each row of the matrix.
YVU420	12	128 (Pi 5), 64 (non-Pi 5)	$(\frac{3}{2} \cdot \text{height}, \text{width})$	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 two rows of V (or U) values on each row of the matrix.
NV12	12	32	Unsupported	YUV 4:2:0 format. A plane of $\text{height} \cdot \text{stride}$ Y values followed by $\text{height}\ /2 \cdot \text{stride}$ interleaved U and V values.
NV21	12	32	Unsupported	YUV 4:2:0 format. A plane of $\text{height} \cdot \text{stride}$ Y values followed by $\text{height}\ /2 \cdot \text{stride}$ interleaved V and U values.
YUYV	16	32	Unsupported	YUV 4:2:2 format. A plane of $\text{height} \cdot \text{stride}$ interleaved values in the order Y U Y V for every two pixels.
YVYU	16	32	Unsupported	YUV 4:2:2 format. A plane of $\text{height} \cdot \text{stride}$ interleaved values in the order Y V Y U for every two pixels.
UYVY	16	32	Unsupported	YUV 4:2:2 format. A plane of $\text{height} \cdot \text{stride}$ interleaved values in the order U Y V Y for every two pixels.
VYUY	16	32	Unsupported	YUV 4:2:2 format. A plane of $\text{height} \cdot \text{stride}$ interleaved values in the order V Y U Y for every two pixels.

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

Figure 6.
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
QtGL preview	Yes	No	Yes	No	Yes	No

	XRGB8888 XBGR8888	RGB888 BGR888	YUV420 YVU420	NV12 NV21	YUYV UYVY	YVYU VYUY
Qt preview	Yes, slow	Yes, slow	Requires OpenCV, very slow	No	No	No
DRM preview	Yes	Yes	Yes	No	No	No
Null preview	Yes	Yes	Yes	Yes	Yes	Yes
JPEG encode	Yes	Yes	Yes	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: those which are not specific to any of the main, lores or raw streams.

Table 1.
Global camera configuration parameters

Parameter name	Permitted values	Description

`"use_case"`	`"preview"`
	`"still"`
	`"video"`
		This parameter only exists as an aid to users, to show the intended use for configurations. The `create_preview_configuration()` will create configurations where this is set to `"preview"`; still and video versions work the same way.

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

`"colour_space"`	`Sycc()`
	`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 `Sycc()`. `create_video_configuration()` chooses `Sycc()` 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 four for preview configurations, one for still capture configurations, and six for video. Increasing the number of buffers will tend to lead to fewer frame drops, although this comes 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.

Parameter name	Permitted values	Description

`"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, though the lores stream can also be used if it is defined. For preview and still use cases the value will be set to `None`. This value can always be overridden when an encoder is started.

`"sensor"`	Dictionary containing
	`"output_size"`
	`"bit_depth"`
		When present, this determines the chosen sensor mode, overriding any raw stream parameters (which will be adjusted to match the chosen sensor mode). This parameter is a dictionary containing the sensor `"output_size"`, a `(width, height)` tuple, and `"bit_depth"`, which is the number of bits in each pixel sample (10 or 12 for most Raspberry Pi cameras).

`"controls"`	Please refer to Appendix C. Camera controls.	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 2.

Stream-specific configuration parameters

Parameter name	Permitted values	Description
`"format"`	A string describing the image format. Please refer to Figure 5.	The pixel and image format. Please refer to Figure 5 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 `rpicam-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 `rpicam-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

Setting enum control values

Some camera controls have enum values, that is to say, an explicitly enumerated list of acceptable values. To use such values, the correct method would be to import `controls` from `libcamera` and use code like the following:

```
Code
from picamera2 import Picamera2
from libcamera import controls

picam2 = Picamera2()
picam2.set_controls({"AeMeteringMode": controls.AeMeteringModeEnum.Spot})
```

Controls that change with the camera configuration

Some controls have limits that change with the camera configuration. In this case the valid range of values can be queried from the `camera_controls` property where each control has a `(minimum_value, maximum_value, default_value)` triple. The controls specifically affected are: `AnalogueGain`, `ExposureTime`, `FrameDurationLimits` and `ScalerCrop`. For example:

```
Code
from picamera2 import Picamera2

picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration())
min_exp, max_exp, default_exp = picam2.camera_controls["ExposureTime"]
```

Camera controls and image metadata

Note that controls also double up as values reported in captured image metadata. Some controls only appear in such metadata, which means they cannot be set, therefore making the read-only. We indicate this in the table below.

Table 3.
Available camera controls.

Control name	Description	Permitted values
`"AeConstraintMode"`	Sets the constraint mode of the AEC/AGC[1]	`AeConstraintModeEnum` followed by one of: `Normal` - normal metering `Highlight` - meter for highlights `Shadows` - meter for shadows `Custom` - user-defined metering
`"AeEnable"`	Allow the AEC/AGC algorithm to be turned on and off. When if is off, there will be no automatic updates to the camera's gain or exposure settings.	`False` - turn AEC/AGC off `True` - turn AEC/AGC on
`"AeExposureMode"`	Sets the exposure mode of the AEC/AGC[1]	`AeExposureModeEnum` followed by one of: `Normal` - normal exposures `Short` - use shorter exposures `Long` - use longer exposures `Custom` - use custom exposures
`"AeFlickerMode"`	Sets the flicker avoidance mode of the AEC/AGC[1]	`AeFlickereModeEnum` followed by one of: `FlickerOff` - no flicker avoidance `FlickerManual` - flicker is avoided with a period set in the `AeFlickerPeriod` control
`"AeFlickerPeriod"`	Sets the lighting flicker period in microseconds[1]	The period of the lighting cycle in microseconds. For example, for 50Hz mains lighting the flicker occurs at

[1] See https://datasheets.raspberrypi.com/camera/raspberry-pi-camera-guide.pdf section 5.8.

Control name	Description	Permitted values
		100Hz, so the period would be 10000 microseconds.
`"AeMeteringMode"`	Sets the metering mode of the AEC/AGC[1]	`AeMeteringModeEnum` followed by one of: `CentreWeighted` - centre weighted metering `Spot` - spot metering `Matrix` - matrix metering `Custom` - custom metering
`"AfMetering"`	Where focus should be measured.	`AfMeteringEnum` followed by one of: `Auto` - use central region of image `Windows` - use the windows given in the `"AfWindows"` control
`"AfMode"`	The autofocus mode.	`AfModeEnum` followed by one of: `Manual` - manual mode, lens position can be set `Auto` - auto mode, can be triggered by application `Continuous` - continuous, runs continuously
`"AfPause"`	Pause continuous autofocus. Only has any effect when in continuous autofocus mode.	`AfPauseEnum` followed by one of: `Deferred` - pause continuous autofocus when no longer scanning `Immediate` - pause continuous autofocus immediately, even if scanning `Resume` - resume continuous autofocus
`"AfRange"`	Range of lens positions to search.	`AfRangeEnum` followed by one of: `Normal` - search the normal range (may exclude very closest objects) `Macro` - search only the closest part of the range `Full` - search everything
`"AfSpeed"`	Speed of the autofocus search.	`AfSpeedEnum` followed by one of: `Normal` - normal speed `Fast` - try to move more quickly
`"AfTrigger"`	Start an autofocus cycle. Only has any effect when in auto mode.	`AfTriggerEnum` followed by one of: `Start` - start the cycle `Cancel` - cancel an in progress cycle
`"AfWindows"`	Location of the windows in the image to use to measure focus.	A list of rectangles (tuples of 4 numbers denoting *x_offset*, *y_offset*, *width* and *height*). The rectangle units refer to the maximum scaler crop window (please refer to the `ScalerCropMaximum` value in the `camera_properties` property).
`"AnalogueGain"`	Analogue gain applied by the sensor.	Consult the `camera_controls` property.
`"AwbEnable"`	Turn the auto white balance (AWB) algorithm on or off. When it is off, there will be no automatic updates to the colour gains.	`False` - turn AWB off `True` - turn AWB on
`"AwbMode"`	Sets the mode of the AWB algorithm[2]	`AwbModeEnum` followed by one of: `Auto` - any illumant `Tungsten` - tungsten lighting

[2] See https://datasheets.raspberrypi.com/camera/raspberry-pi-camera-guide.pdf section 5.7.

Control name	Description	Permitted values
		`Fluorescent` - fluorescent lighting `Indoor` - indoor illumination `Daylight` - daylight illumination `Cloudy` - cloudy illumination `Custom` - custom setting
`"Brightness"`	Adjusts the image brightness where −1.0 is very dark, 1.0 is very bright, and 0.0 is the default "normal" brightness.	Floating point number from −1.0 to 1.0
`"ColourCorrectionMatrix"`	The $3 \cdot 3$ matrix used within the image signal processor (ISP) to convert the raw camera colours to sRGB. This control appears only in captured image metadata and is read-only.	Tuple of nine floating point numbers between −16.0 and 16.0.
`"ColourGains"`	Pair of numbers where the first is the red gain (the gain applied to red pixels by the AWB algorithm) and the second is the blue gain. Setting these numbers disables AWB.	Tuple of two floating point numbers between 0.0 and 32.0.
`"ColourTemperature"`	An estimate of the colour temperature (in Kelvin) of the current image. It is only available in captured image metadata, and is read-only.	Integer
`"Contrast"`	Sets the contrast of the image, where zero means "no contrast", 1.0 is the default "normal" contrast, and larger values increase the contrast proportionately.	Floating point number from 0.0 to 32.0
`"DigitalGain"`	The amount of digital gain applied to an image. Digital gain is used automatically when the sensor's analogue gain control cannot go high enough, and so this value is only reported in captured image metadata. It cannot be set directly - users should set the `AnalogueGain` instead and digital gain will be used when needed.	Floating point number
`"ExposureTime"`	Exposure time for the sensor to use, measured in microseconds.	Consult the `camera_controls` property
`"ExposureValue"`	Exposure compensation value in "stops", which adjusts the target of the AEC/AGC algorithm. Positive values increase the target brightness, and negative values decrease it. Zero represents the base or "normal" exposure level.	Floating point number between −8.0 and 8.0
`"FrameDuration"`	The amount of time (in microseconds) since the previous camera frame. This value is only available in captured image metadata and is read-only. To change the camera's framerate, the "FrameDurationLimits" control should be used.	Integer
`"FrameDurationLimits"`	The maximum and minimum time that the sensor can take to deliver a frame, measured in microseconds. So the reciprocals of these values (first divided by 1000000) will give the minimum and maximum framerates that the sensor can deliver.	Consult the `camera_controls` property
`"HdrChannel"`	Reports which HDR channel the current frame represents. It is read-only and cannot be set.	`HdrChannelEnum` followed by one of: `HdrChannelNone` - image not used for HDR

Control name	Description	Permitted values
		`HdrChannelShort` - a short exposure image for HDR `HdrChannelMedium` - a medium exposure image for HDR `HdrChannelLong` - a long exposure image for HDR
`"HdrMode"`	Whether to run the camera in an HDR mode (distinct from the in-camera HDR supported by the Camera Module 3). Most of these HDR features work only on Pi 5 or later devices.	`HdrModeEnum` followed by one of: `Off` - disable HDR (default) `SingleExposure` - combine multiple short exposure images, this is the recommended mode (Pi 5 only) `MultiExposure` - combine short and long images, ony recommended when a scene is completely static (Pi 5 only) `Night` - an HDR mode that combines multiple low light images, and can recover some highlights (Pi 5 only) `MultiExposureUnmerged` - return unmerged distinct short and long exposure images.
`"LensPosition"`	Position of the lens. The units are dioptres (reciprocal of the distance in metres).	Consult the `camera_controls` property.
`"Lux"`	An estimate of the brightness (in lux) of the scene. It is available only in captured image metadata and is read-only.	Integer
`"NoiseReductionMode"`	Selects a suitable noise reduction mode. Normally Picamera2's configuration will select an appropriate mode automatically, so it should not normally be necessary to change it. The `HighQuality` noise reduction mode can be expected to affect the maximum achievable framerate.	`draft.NoiseReductionModeEnum` followed by one of: `Off` - no noise reduction `Fast` - fast noise reduction `HighQuality` - best noise reduction
`"Saturation"`	Amount of colour saturation, where zero produces greyscale images, 1.0 represents default “normal” saturation, and higher values produce more saturated colours.	Floating point number from 0.0 to 32.0
`"ScalerCrop"`	The scaler crop rectangle determines which part of the image received from the sensor is cropped and then scaled to produce an output image of the correct size. It can be used to implement digital pan and zoom. The coordinates are always given from within the full sensor resolution.	A `libcamera.Rectangle` consisting of: x_offset y_offset width height
`"SensorTimestamp"`	The time this frame was produced by the sensor, measured in nanoseconds since the system booted. The time is sampled on the camera start of frame interrupt, which occurs as the first pixel of the new frame is written out by the sensor. This control appears only in captured image metadata and is read-only.	Integer
`"SensorBlackLevels"`	The black levels of the raw sensor image. This control appears only in captured image metadata and is read-only. One value is reported for each of the four Bayer channels, scaled up as if the full pixel	Tuple of four integers

Control name	Description	Permitted values
	range were 16 bits (so 4096 represents a black level of 64 in 10-bit raw data).	
`"Sharpness"`	Sets the image sharpness, where zero implies no additional sharpening is performed, 1.0 is the default “normal” level of sharpening, and larger values apply proportionately stronger sharpening.	Floating point number from 0.0 to 16.0
`"SyncMode"`	For synchronised capture from multiple cameras. Sets the device into either “server” or “client” synchronisation mode. Please refer to the synchronised capture in Section 6.4.3. Capturing Synchronised Still Images with Multiple Cameras for more information.	`rpi.SyncModeEnum` followed by one of: `Off` - turn off sync mode `Server` - enable sync mode and act as *server* `Client` - enable sync mode and act as *client*
`"SyncReady"`	When in sync mode, indicates to the application the first frame for which the cameras are synchronised. Applications should wait for this value to be non-zero before using frames. Before this point, frames may not be synchronised.	Integer
`"SyncTimer"`	When cameras are in sync mode, this value represents the estimated time in microseconds until the “synchronisation point” at which the “SyncReady” control will become non-zero. After this point, the value reported here will be negative (as the time has elapsed).	Integer
`"SyncFrames"`	When cameras are in sync mode, this value can be set for the *server* only as the delay (in frames) before all cameras will be flagged as being synchronised (by `"SyncReady"` becoming non-zero). The delay here must be long enough for all the clients to have been started, and to have had some time to adjust to the server’s timing messages.	Positive integer

Appendix D. Camera properties

All the available camera properties are listed in the table below. Some properties are updated only when a camera mode is configured. They can be accessed through the Picamera2 object's `camera_properties` property and are all read-only.

Table 4.

Camera properties

Property name	Description	Datatype
`"ColourFilterArrangement"`	A number representing the native Bayer order of sensor (before any rotation is taken into account).	`0` - RGGB `1` - GRBG `2` - GBRG `3` - BGGR `4` - RGB (non-Bayer) `5` - monochrome
`"Location"`	An integer which specifies where on the device the camera is situated (for example, front or back). For the Raspberry Pi, the value has no meaning.	Integer
`"Model"`	The name that the attached sensor advertises.	String
`"PixelArrayActiveAreas"`	The active area of the sensor's pixel array within the entire sensor pixel array. Given as a tuple of values: `(x_offset, y_offset, width, height)`	Tuple of four integers
`"PixelArraySize"`	The size of the active pixel area as an `(x, y)` tuple. This is the full available resolution of the sensor.	Tuple of two integers
`"Rotation"`	The rotation of the sensor relative to the camera board. On many Raspberry Pi devices, the sensor is actually upside down when the camera board is held with the connector at the bottom, and these will return a value of 180° here.	Integer
`"ScalerCropMaximum"`	This value is updated when a camera mode is configured. It returns the rectangle as a `(x_offset, y_offset, width, height)` tuple within the pixel area active area, that is read out by this camera mode.	Tuple of 4 integers
`"SensorSensitivity"`	This value is updated when a camera mode is configured. It represents a relative sensitivity of this camera mode compared to other camera modes. Usually, camera modes all have the same sensitivity so that the same exposure time and gain yield an image of the same brightness. Sometimes cameras have modes where this is not true, and to get the same brightness you would have to adjust the total requested exposure by the ratio of these sensitivities. For most sensors this will always return 1.0.	Floating point number
`"UnitCellSize"`	The physical size of this sensor's pixels, if known. Given as an `(x, y)` tuple in units of nanometres.	Tuple of two integers

Raspberry Pi

Raspberry Pi Ltd