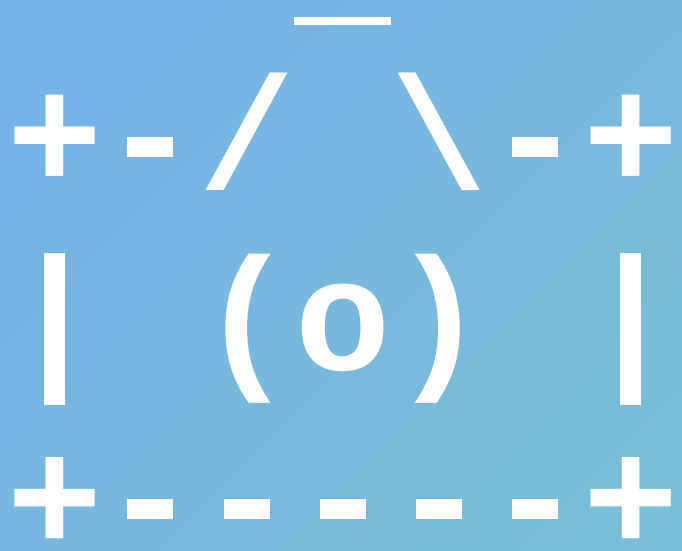


Implementing ISP algorithms in libcamera

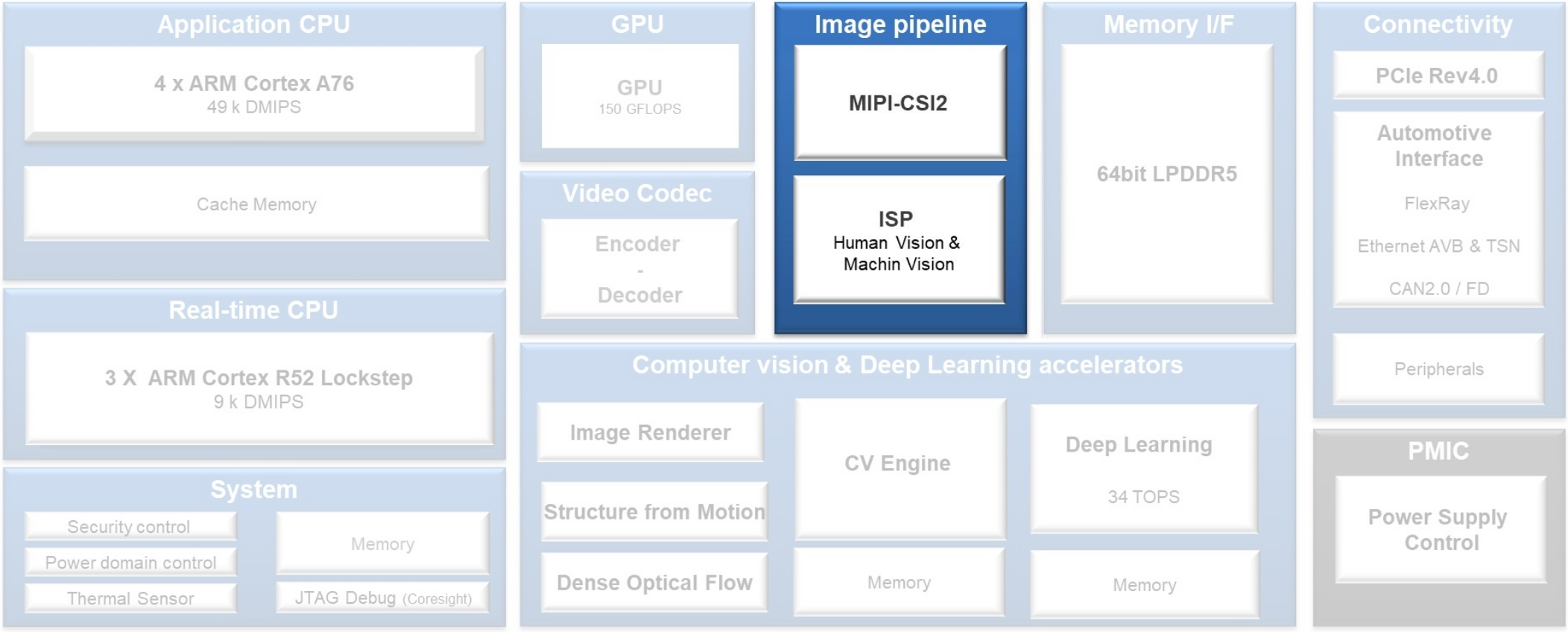
Embedded Recipes 2023
Paris, 2023-09-29

Laurent Pinchart
laurent.pinchart@ideasonboard.com

IDEAS ON BOARD

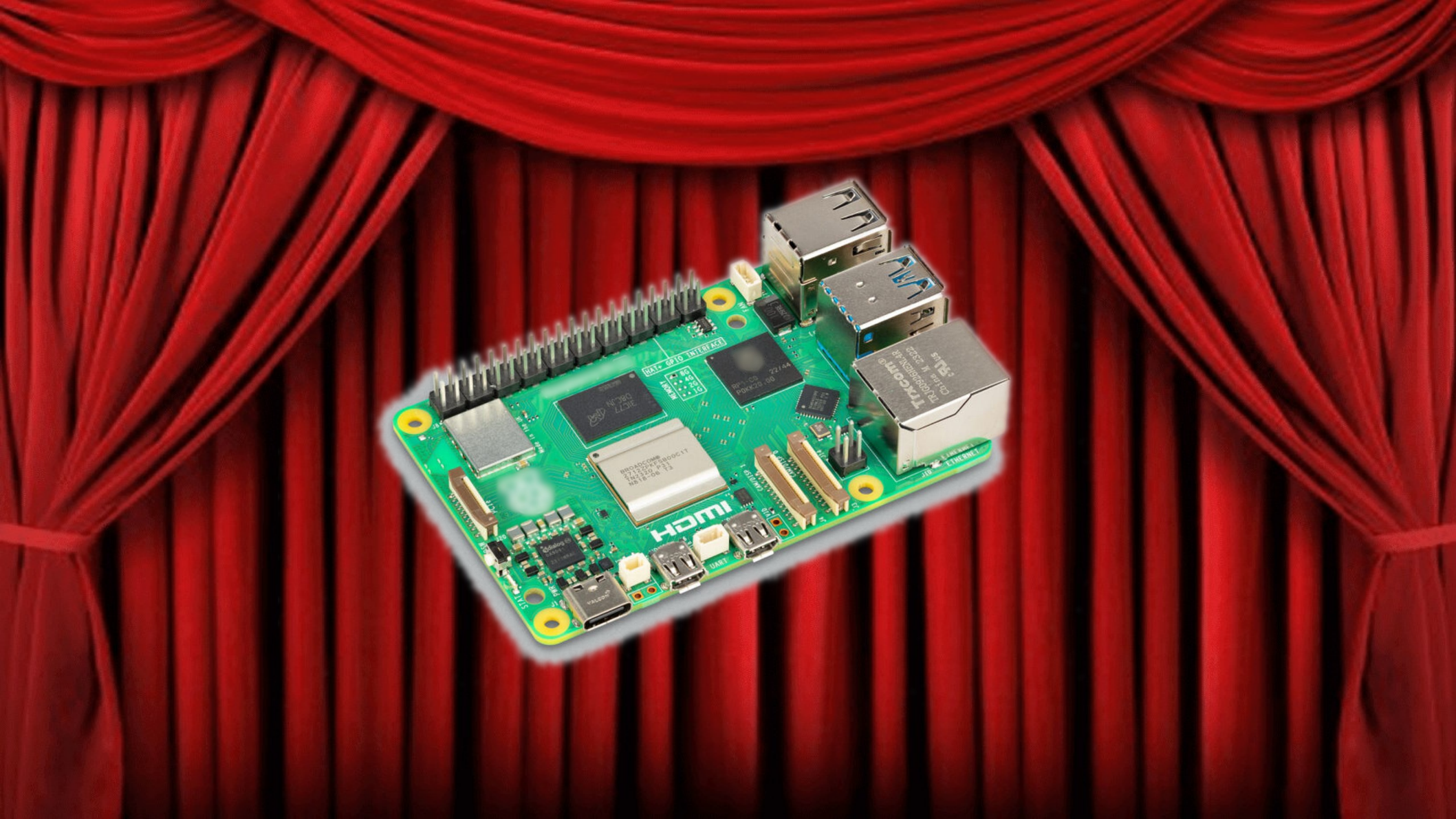
IDEAS ON BOARD

An Amazing SoC



IDEAS ON BOARD

With The Best Camera Hardware


HAT+ GPIO INTERFACE
HDMI
UART

Learn How to Support Your SoC and ISP in Libcamera - Laurent Pinchart, Ideas on Board

The Linux Foundation
170K subscribers

Subscribe

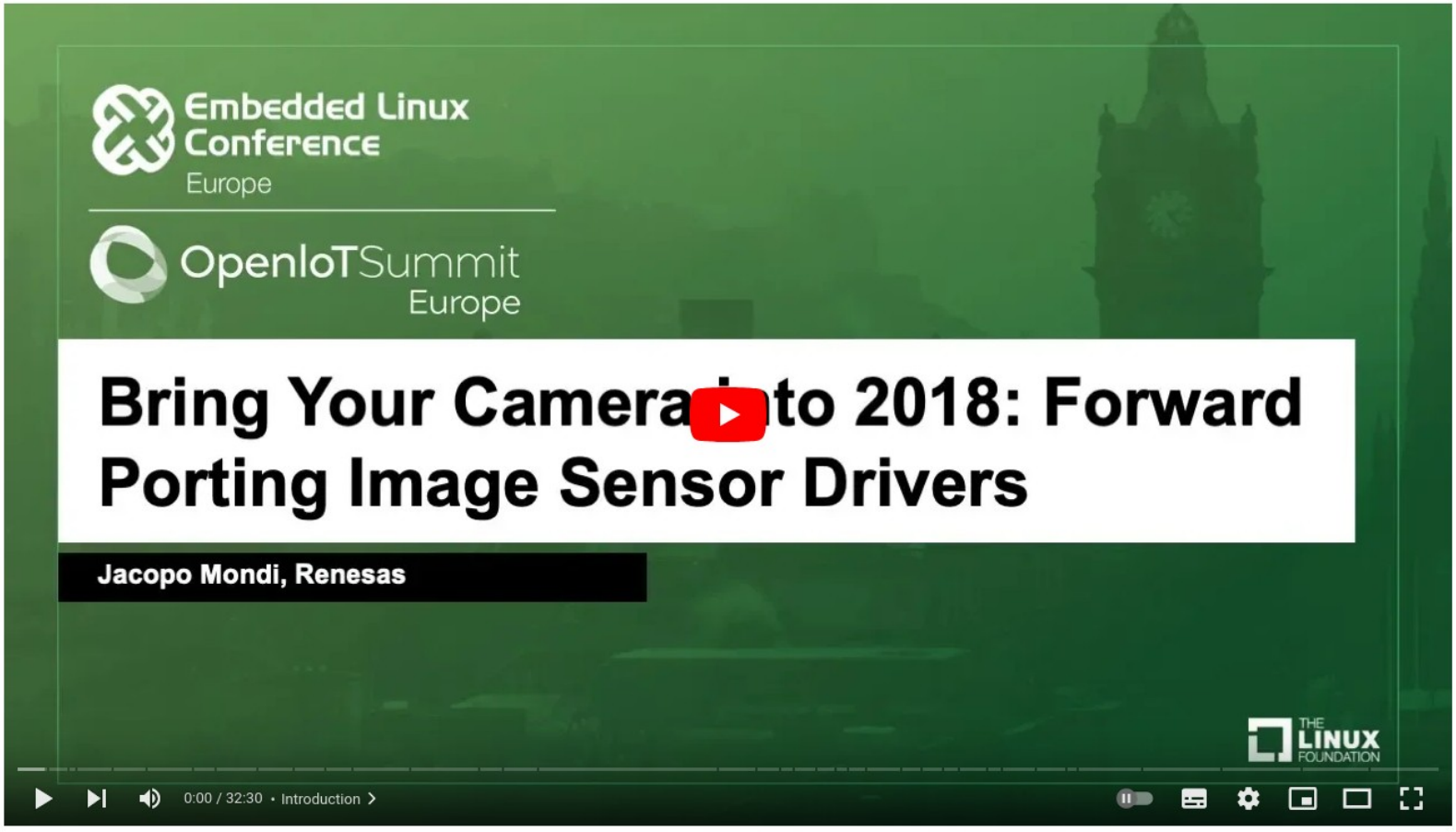
5 Share Save

IDEAS ON BOARD

https://www.youtube.com/watch?v=x9ndjdvXPl0

IDEAS ON BOARD

The Broccoli Pi



Bring Your Camera into 2018: Forward Porting Image Sensor Drivers - Jacopo Mondi

The Linux Foundation
168K subscribers
Subscribe

60 Share Save

IDEAS ON BOARD

https://www.youtube.com/watch?v=PJVlvUf0gP4

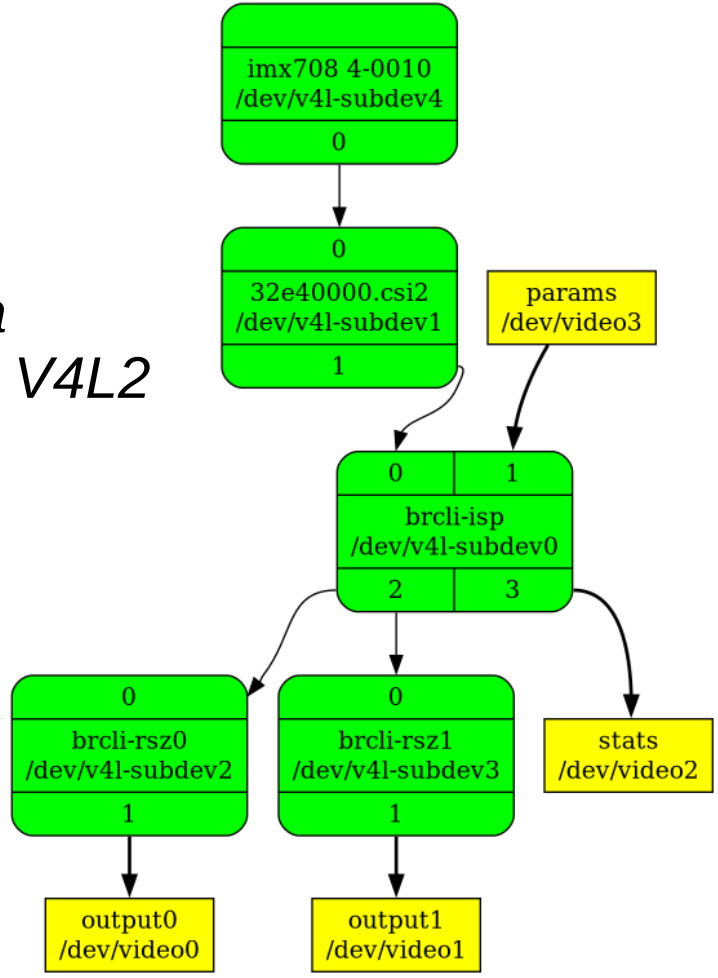
Image Signal Processing (ISP) Drivers & How to Merge One Upstream - Helen Koike, Collabora

The Linux Foundation
168K subscribers
Subscribe
37
Share
Save

IDEAS ON BOARD

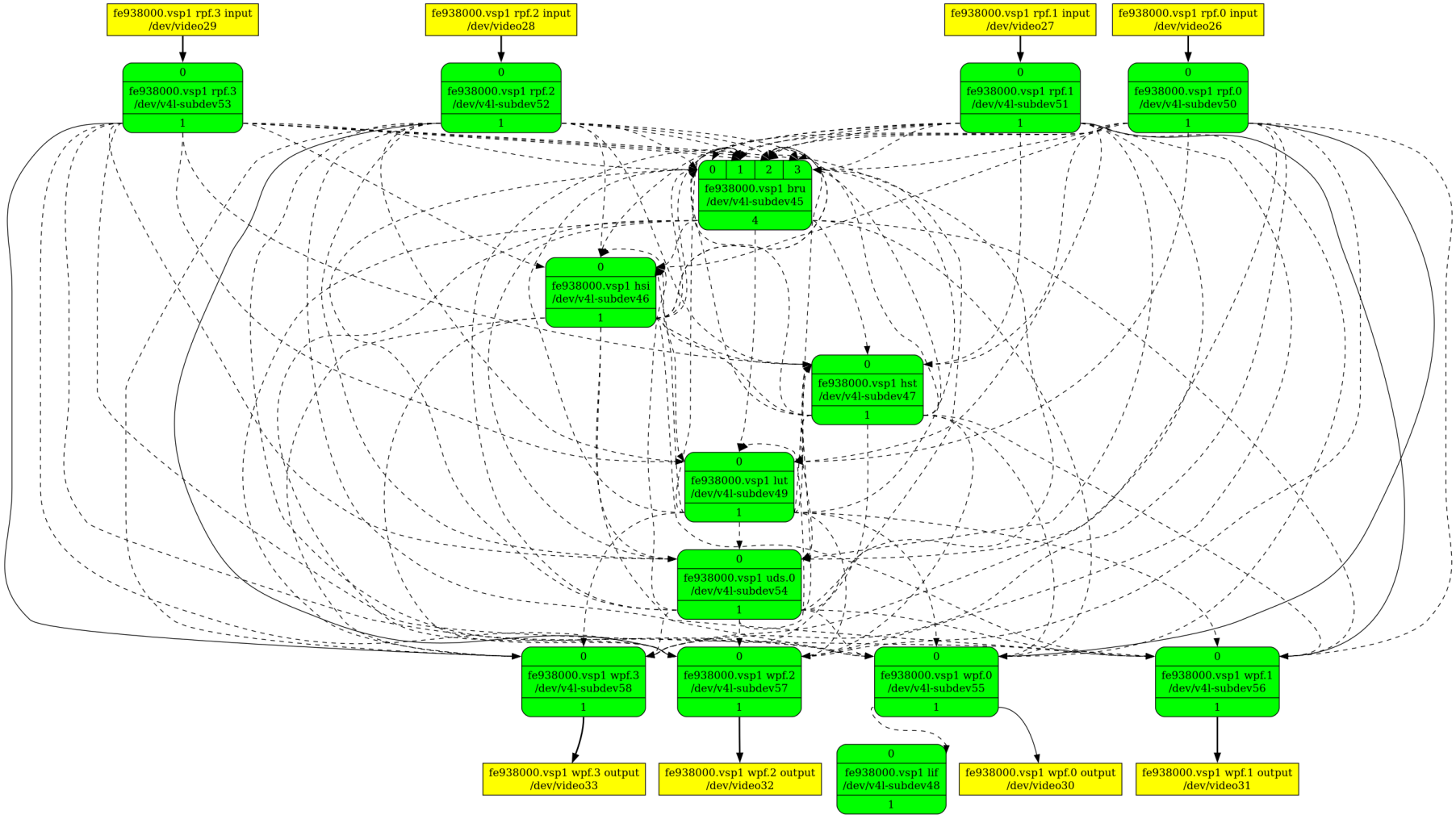
https://www.youtube.com/watch?v=SuAiJOtCxQY

Use the Media Controller and V4L2 APIs



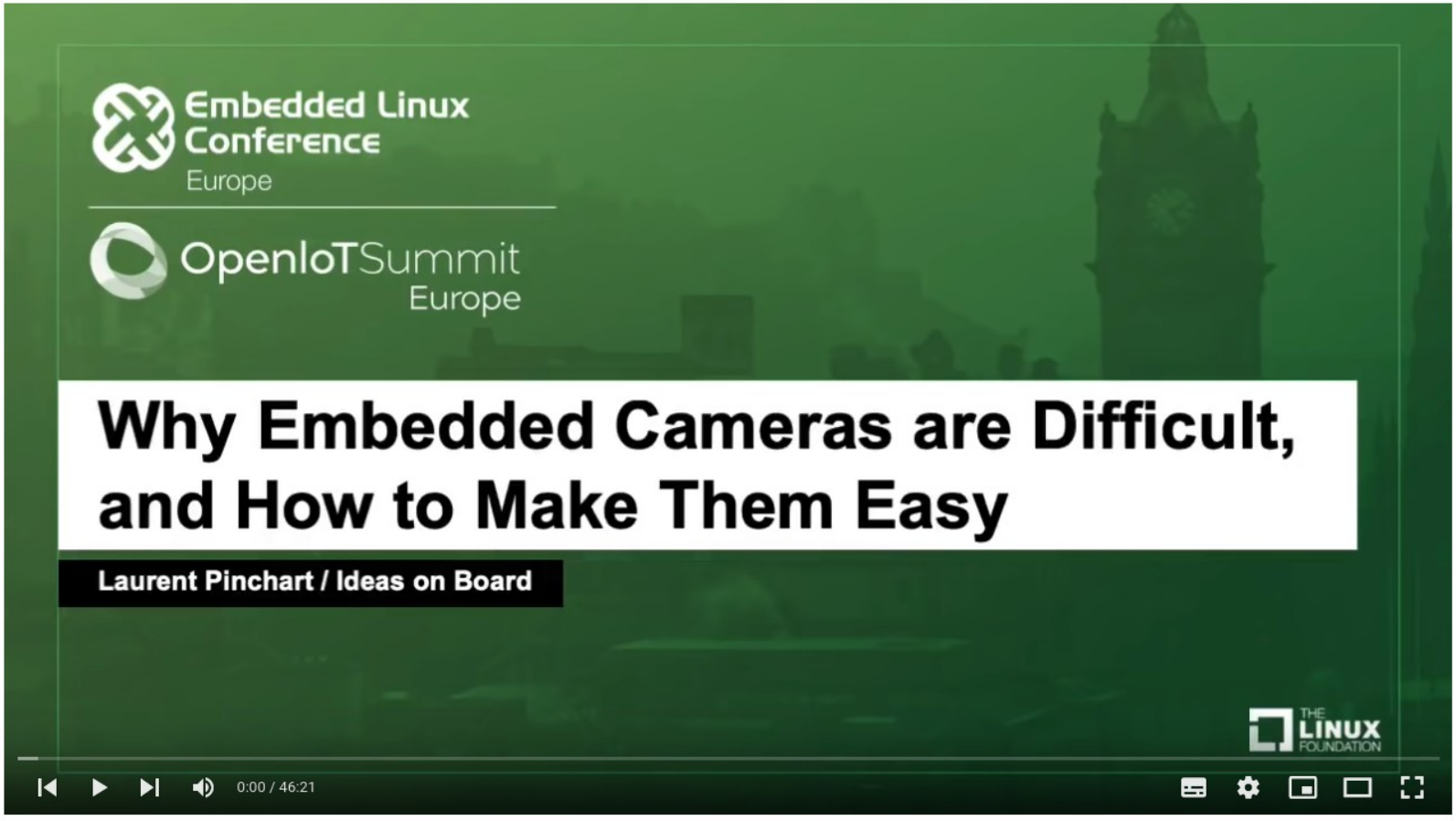
Document and upstream the drivers

The community can provide support



Complex Hardware

A Bitter Tasting Broccoli

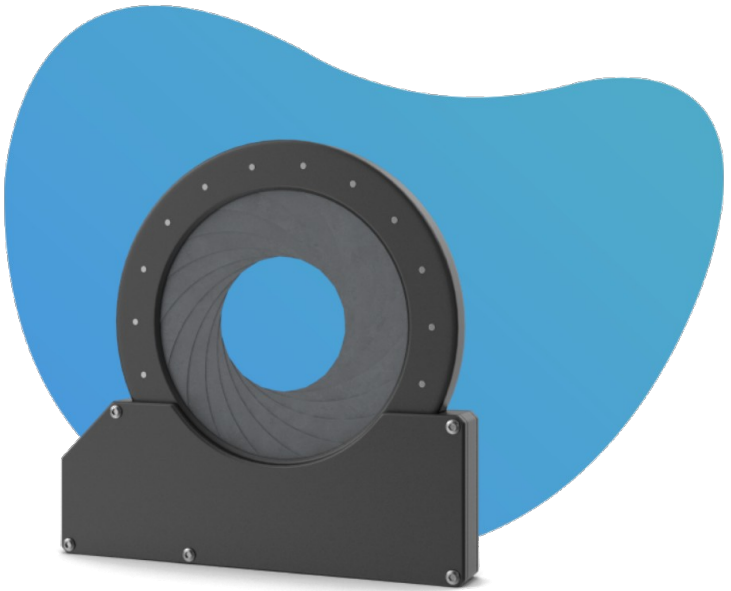


Why Embedded Cameras are Difficult, and How to Make Them Easy - Laurent Pinchart, Ideas on Board

The Linux Foundation
168K subscribers
Subscribe
79
Share
Save

IDEAS ON BOARD

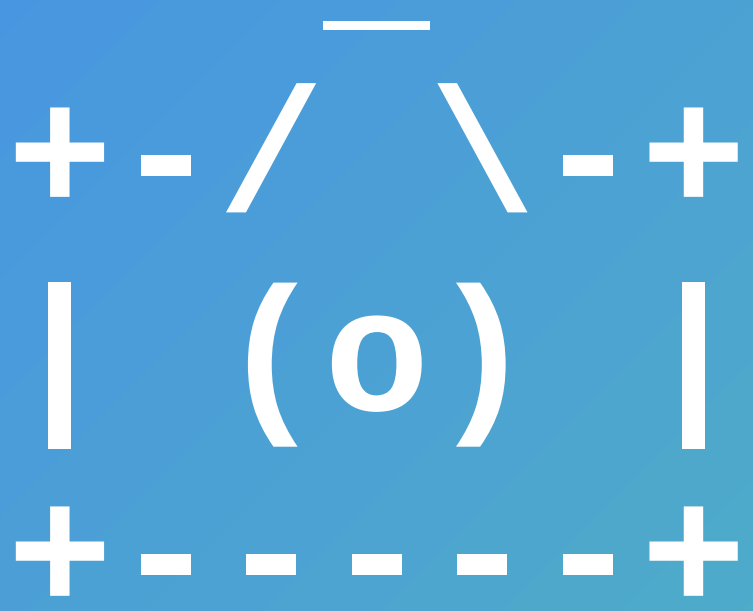
https://www.youtube.com/watch?v=GIhV7tiUji0



Hi, we're libcamera.

An open source camera stack and framework for Linux, Android, and ChromeOS

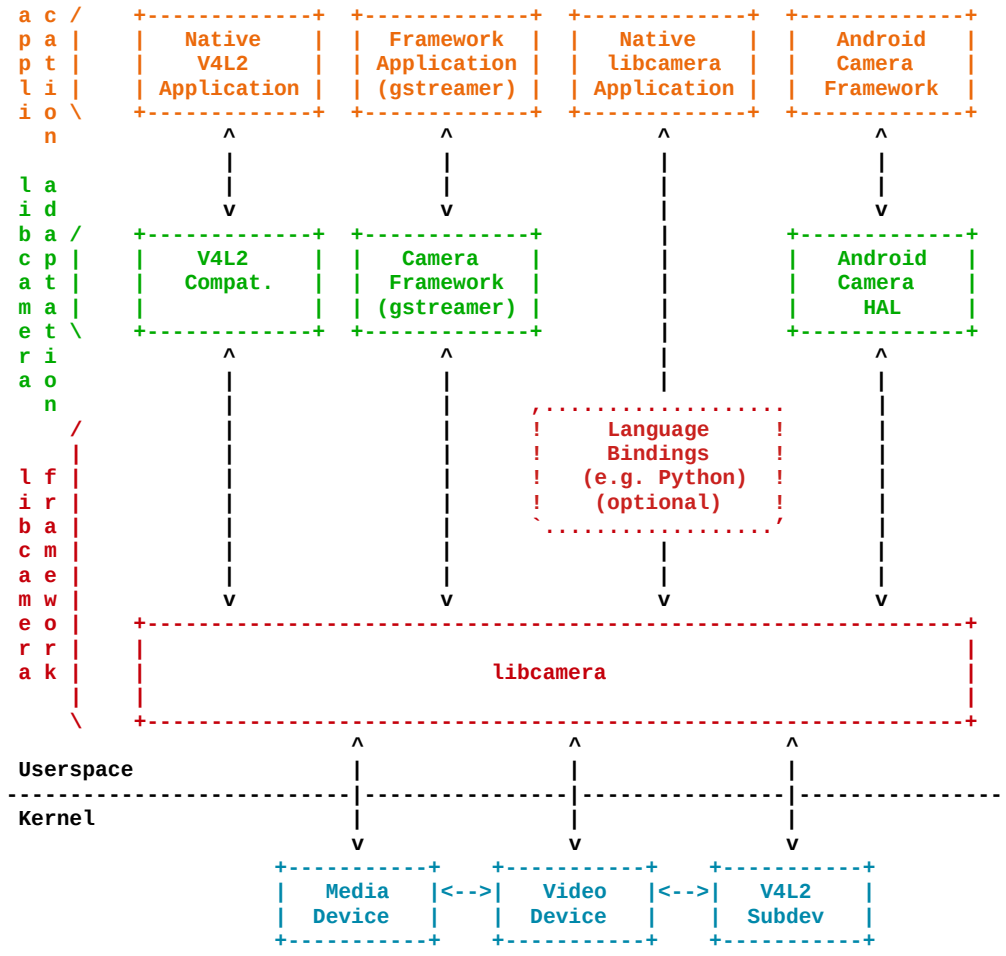
Getting Started



libcamera

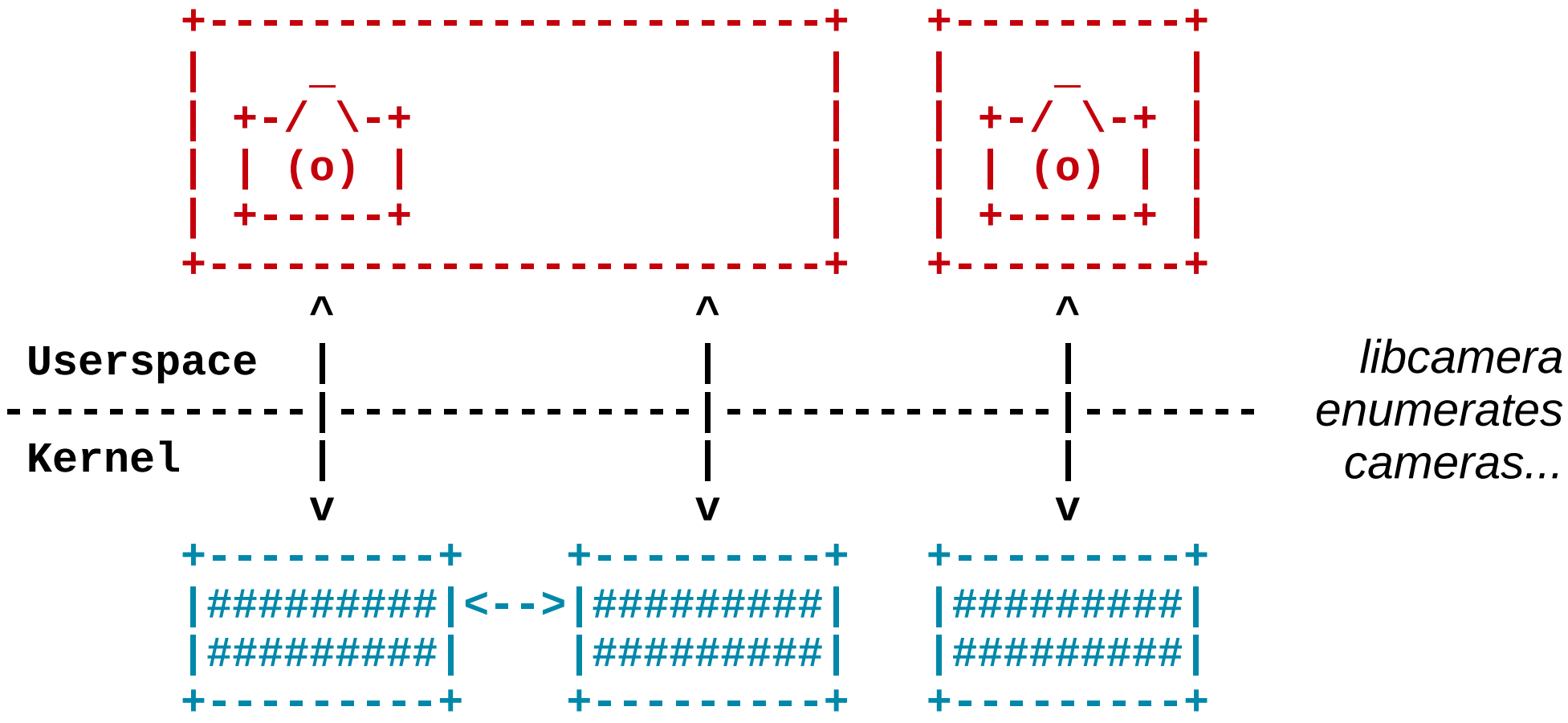
IDEAS ON BOARD

libcamera provides a complete userspace camera stack.

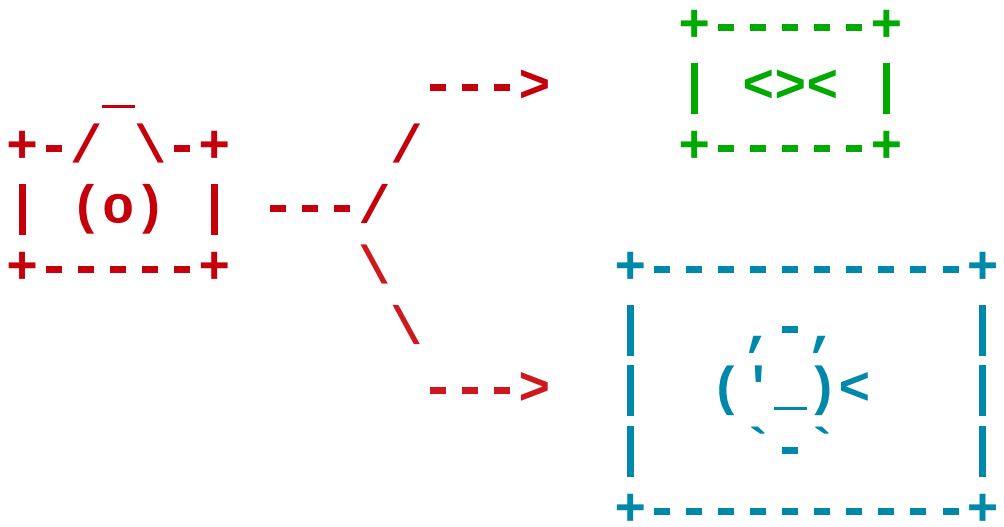


The 'Mesa' of the camera world.

Camera Stack

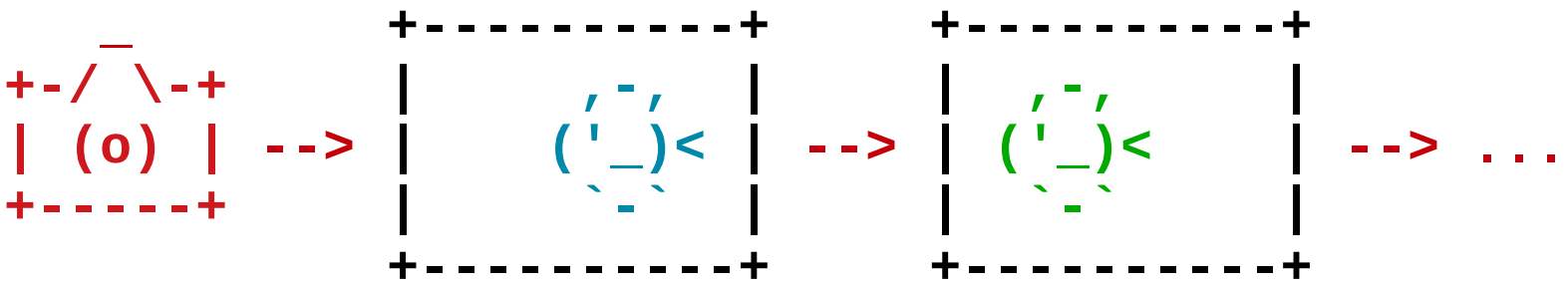


Camera Devices & Enumeration



It supports multiple concurrent streams for the same camera...

Streams

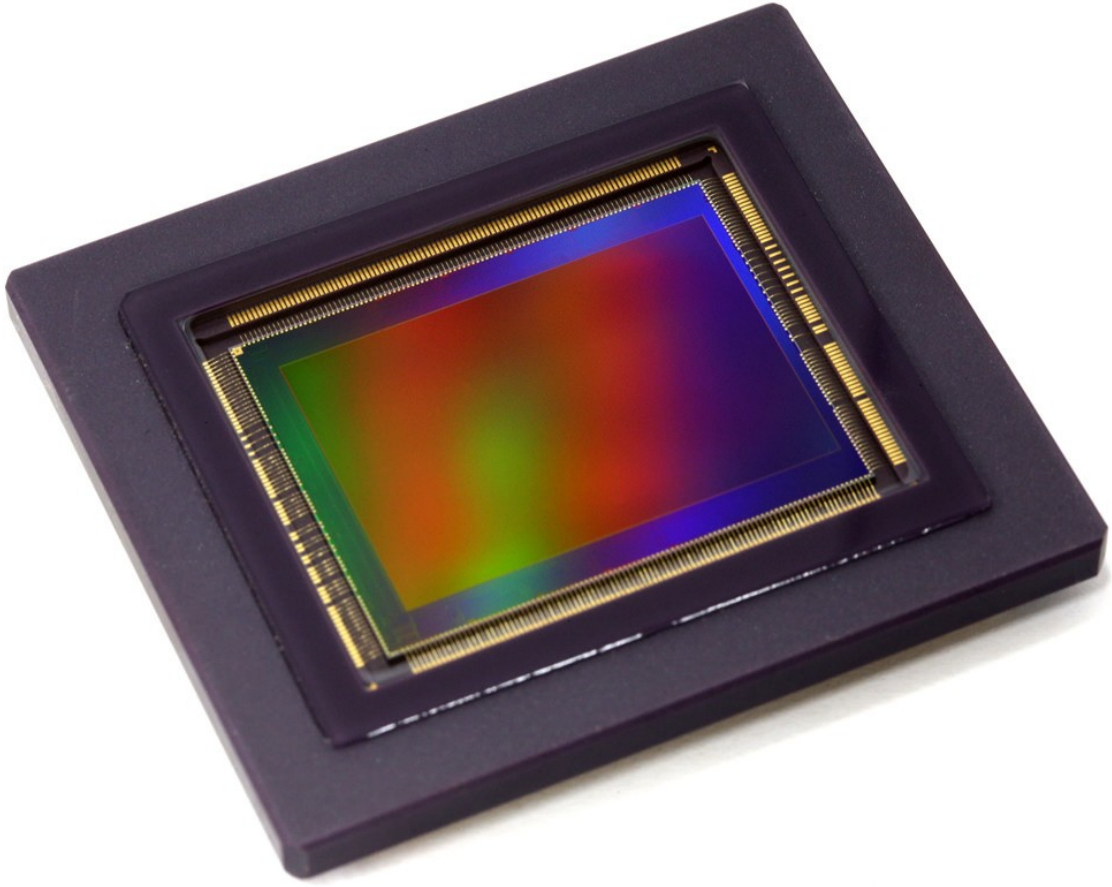


… and per-frame controls.

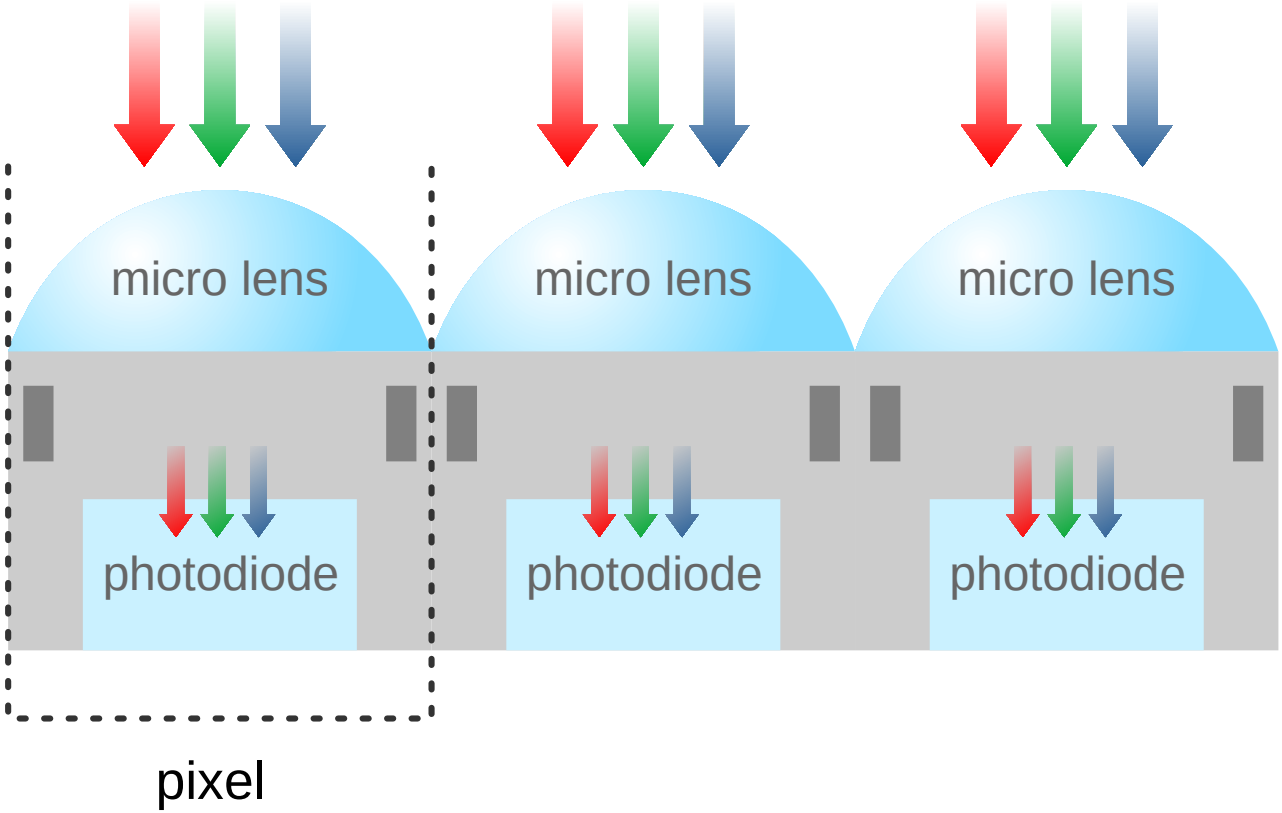
Per-Frame Controls

What is an ISP

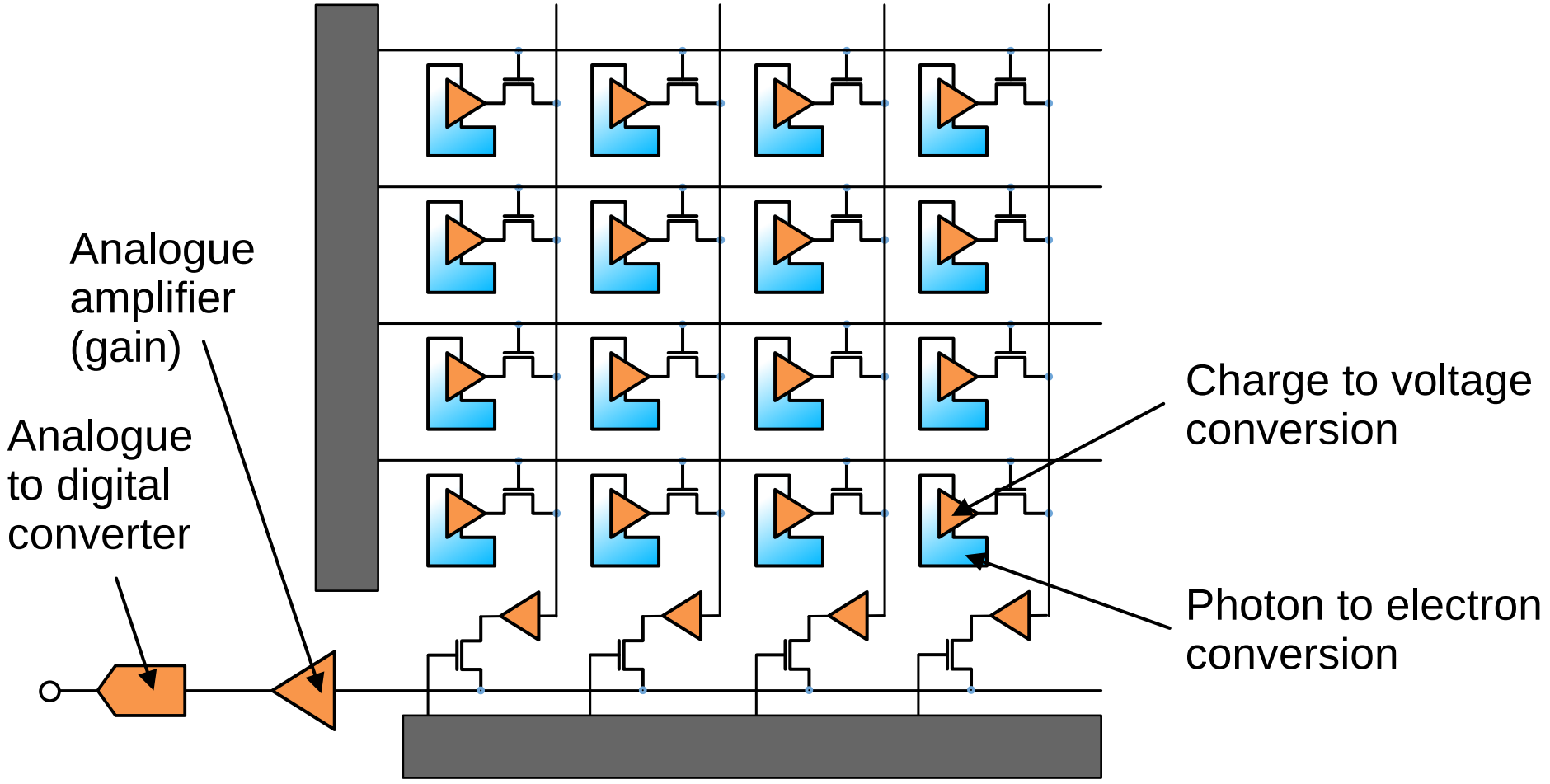
IDEAS ON BOARD



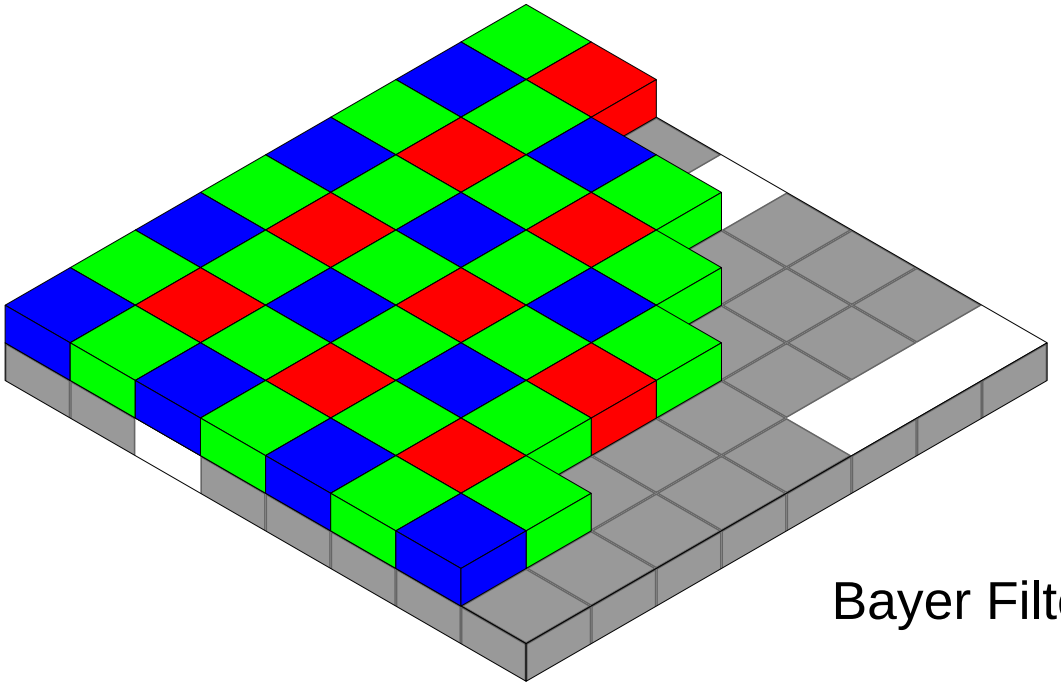
CMOS Sensor


micro lens
micro lens
micro lens
photodiode
photodiode
photodiode
pixel

CMOS Sensor


Analogue amplifier (gain)
Analogue to digital converter
Charge to voltage conversion
Photon to electron conversion

CMOS Sensor



Bayer Filter

Colour Filter Array

source: https://en.wikipedia.org/wiki/Bayer_filter

Original

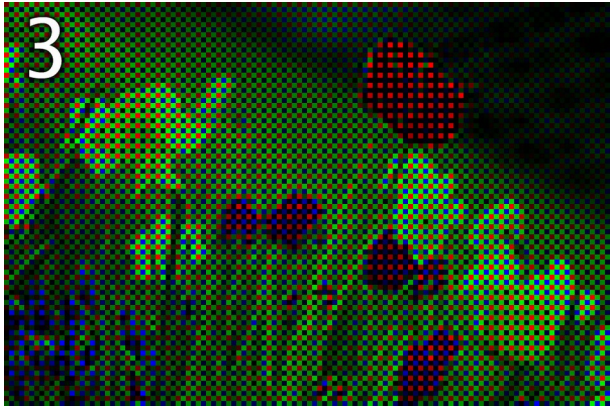
120x80 Bayer image

CFA Interpolation

source: https://en.wikipedia.org/wiki/Bayer_filter

Original

120x80 Bayer image



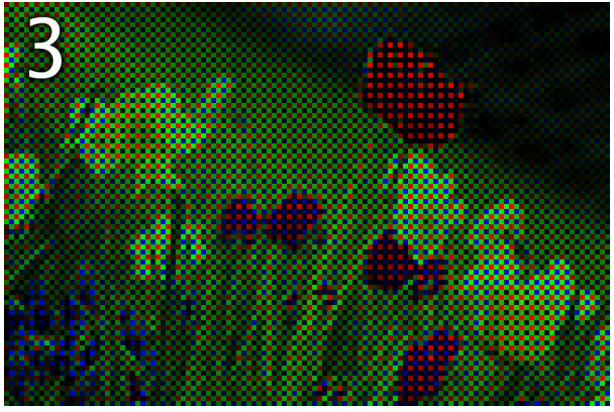
Colour-coded

CFA Interpolation

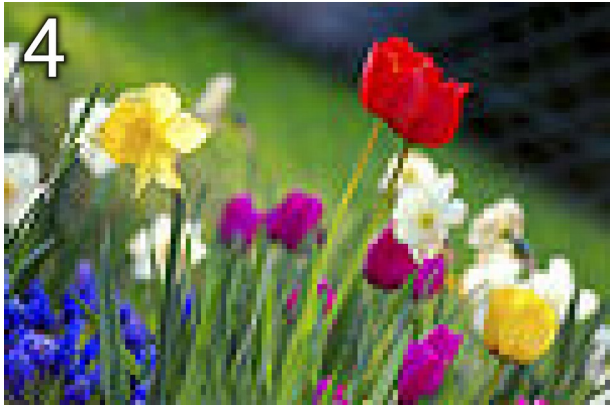
source: https://en.wikipedia.org/wiki/Bayer_filter

Original

120x80 Bayer image



Colour-coded

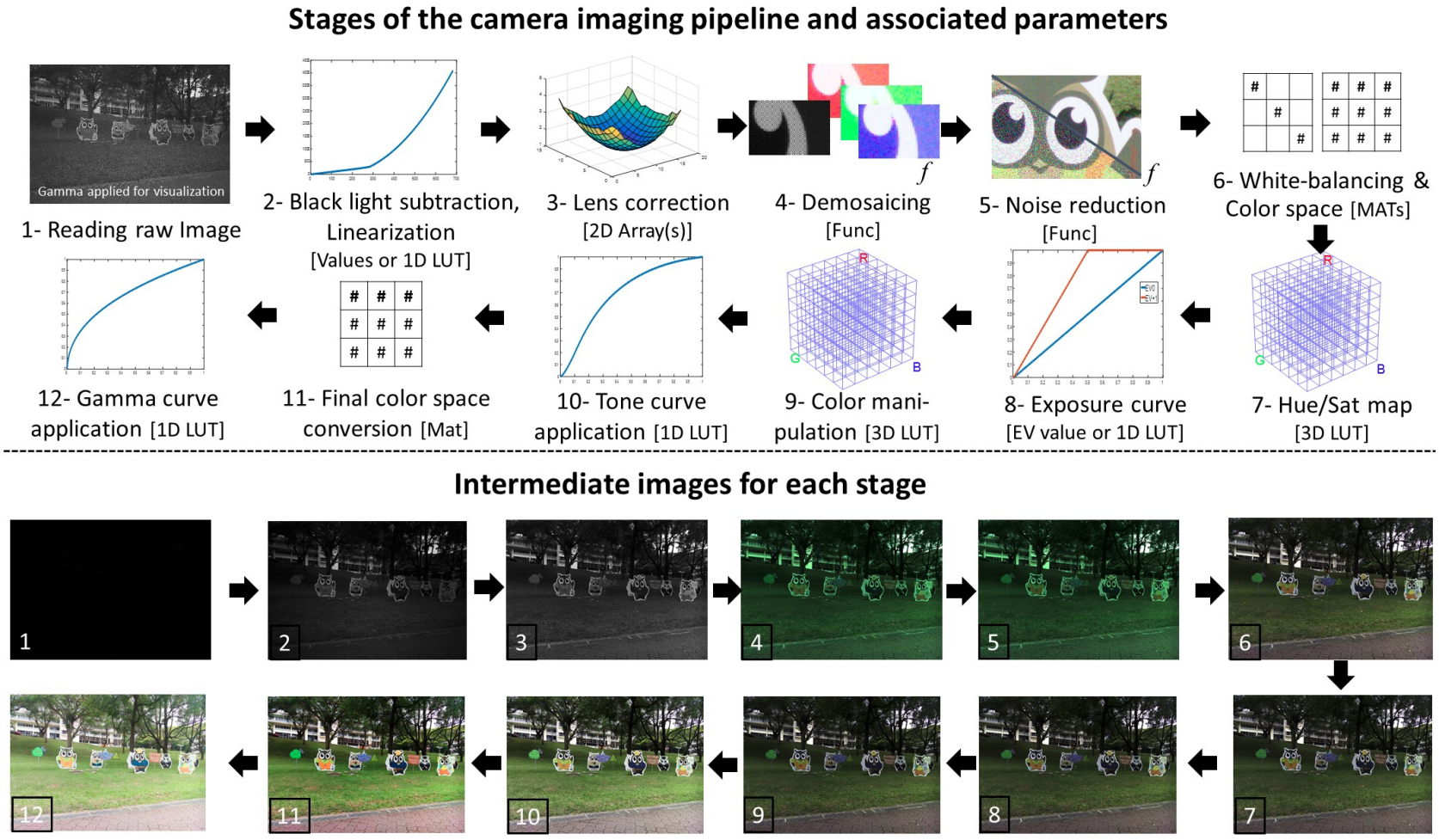


Colour interpolation

CFA Interpolation

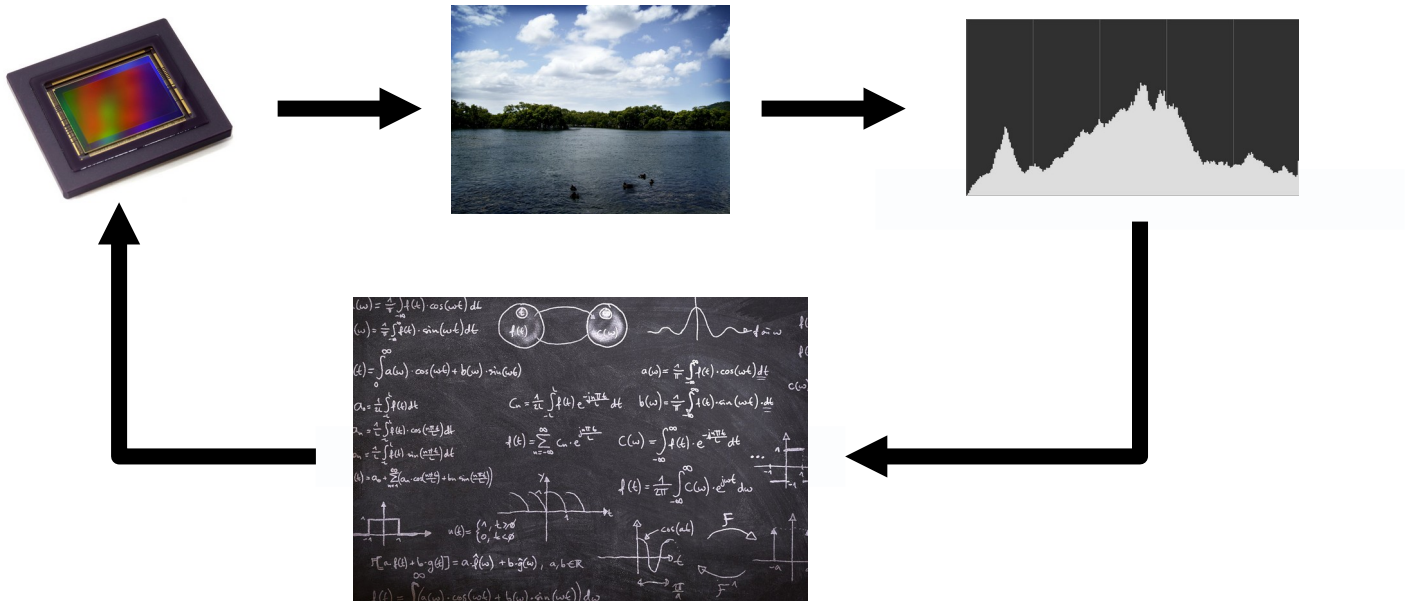
source: https://en.wikipedia.org/wiki/Bayer_filter

Lens Shading



Camera Pipeline

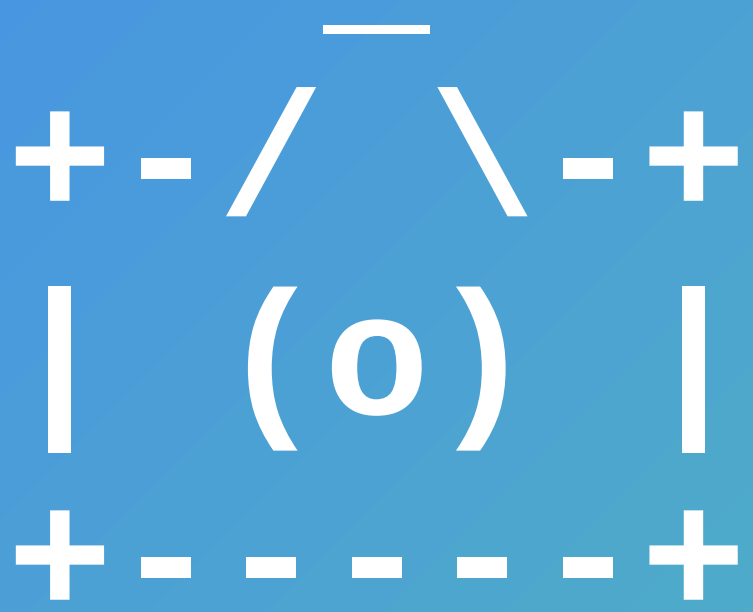
source: https://karaimer.github.io/camera-pipeline/



Auto Algorithms (a.k.a. 3A)

IQ Tuning

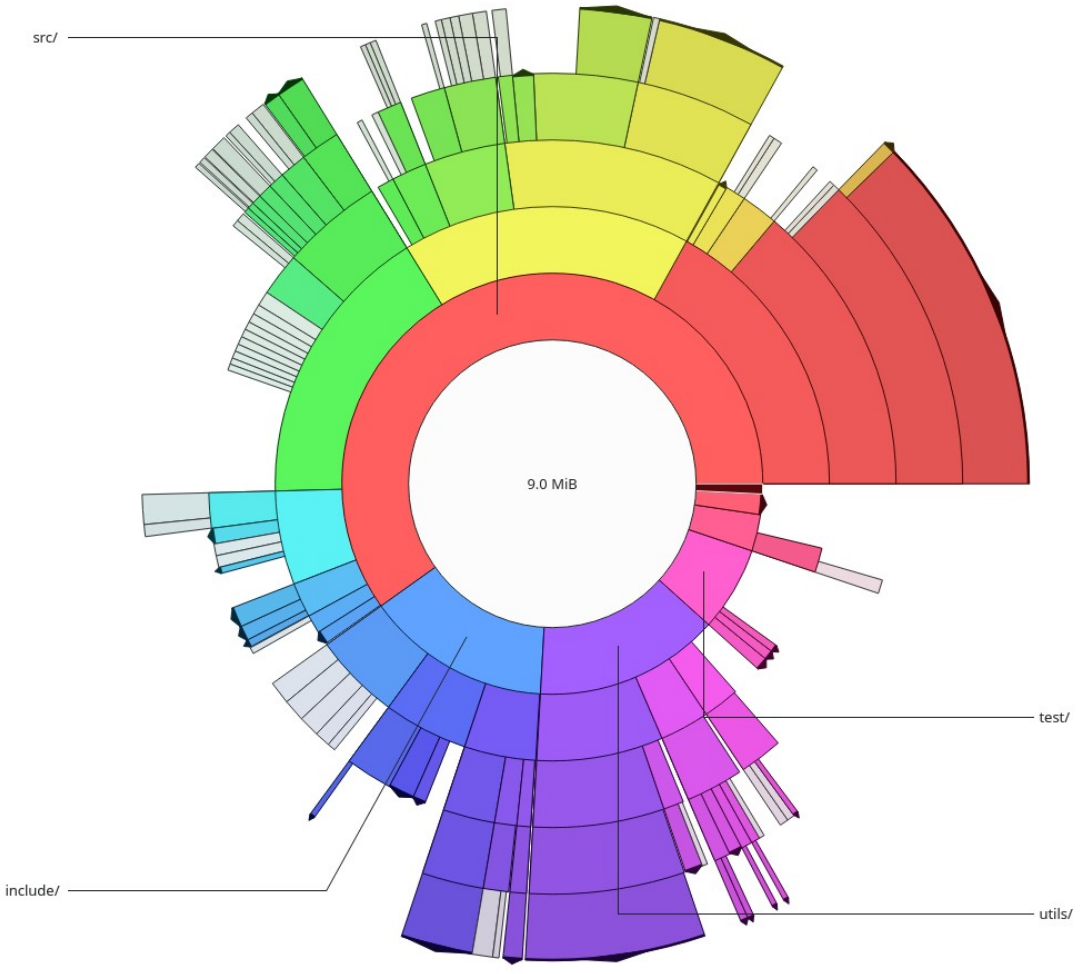
source: https://www.flickr.com/photos/davedugdale/15043975135



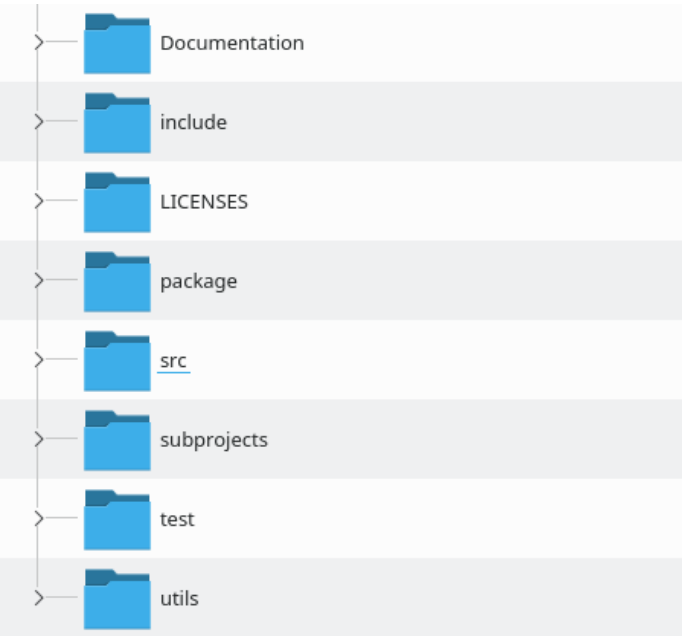
libcamera source tree

(for reference only)

IDEAS ON BOARD



Navigating in libcamera



Documentation, guides, tutorials, …

Headers (public API, internal API, …)

Licenses

Distribution packaging

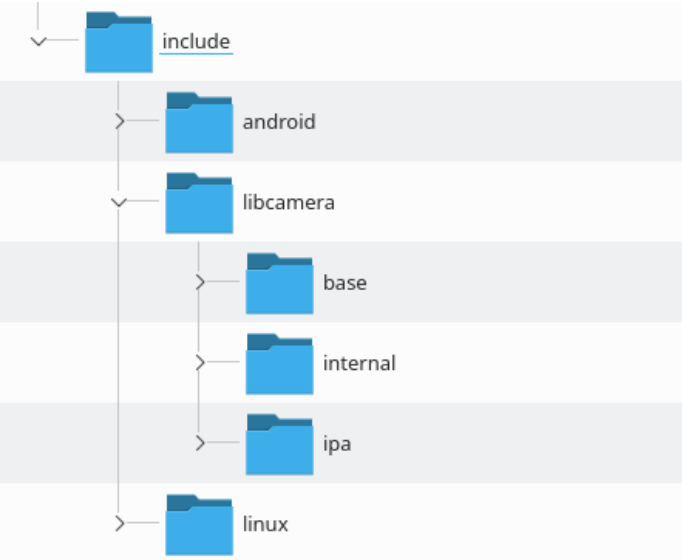
Source

Meson subprojects

Unit tests

Miscellaneous utilities

Project Structure



Android headers copied from Android sources

libcamera public & internal APIs

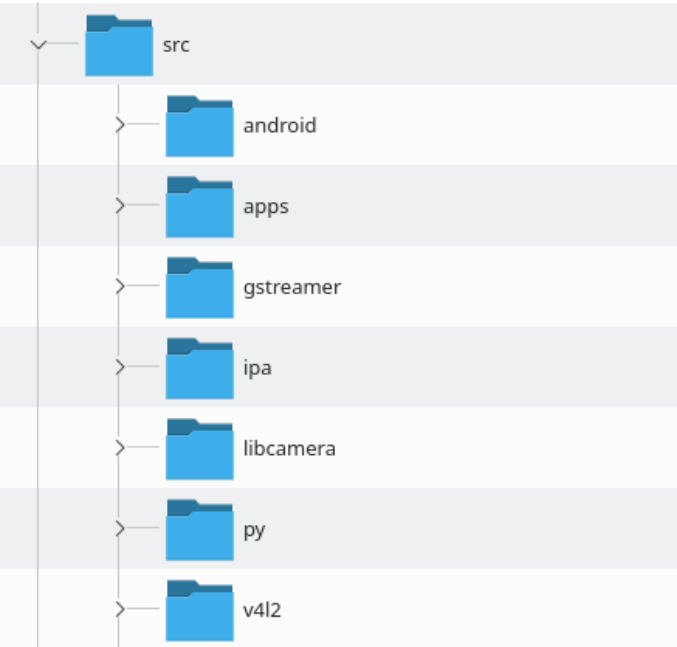
Base API (a.k.a. “NIH boost”)

Internal API

IPA module protocols

Linux kernel headers copied from kernel source

Project Structure – /include



Android camera HAL

test applications (CLI, GUI, compliance test)

GStreamer source element

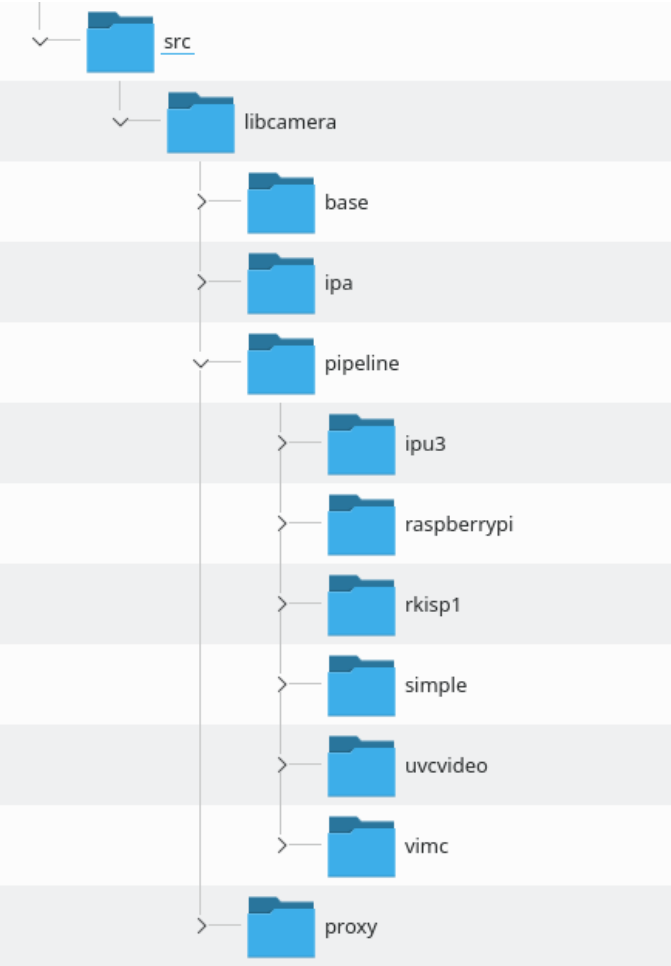
Image Processing Algorithm modules

libcamera core

Python bindings and sample applications

v4l2 compatibility

Project Structure – /src



Base helpers (a.k.a. “NIH boost”)

IPA IPC documentation (placeholder, auto-generated)

Pipeline handlers

Intel IPU3

Raspberry Pi

Rockchip ISP

“simple” pipeline handler

USB Video Class (USB webcams)

“vimc” (Virtual Media Controller)

IPA IPC code (placeholder, auto-generated)

Project Structure – /src/libcamera

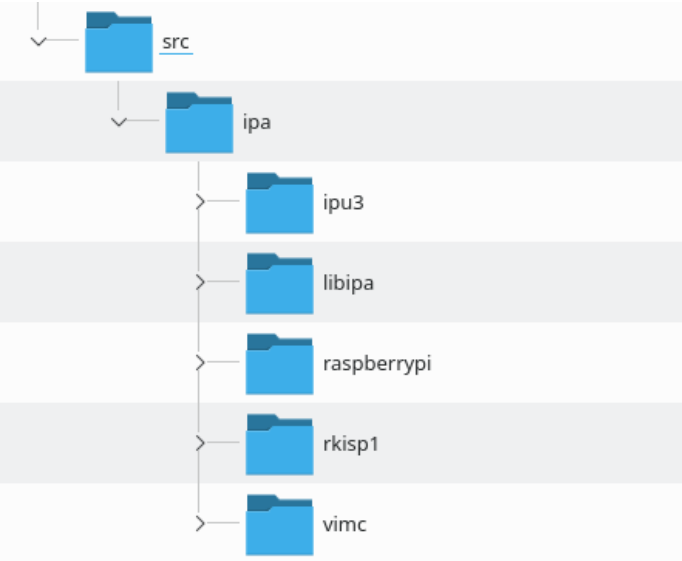


Image Processing Algorithm sources

Intel IPU3

Shared IPA helpers

Raspberry Pi

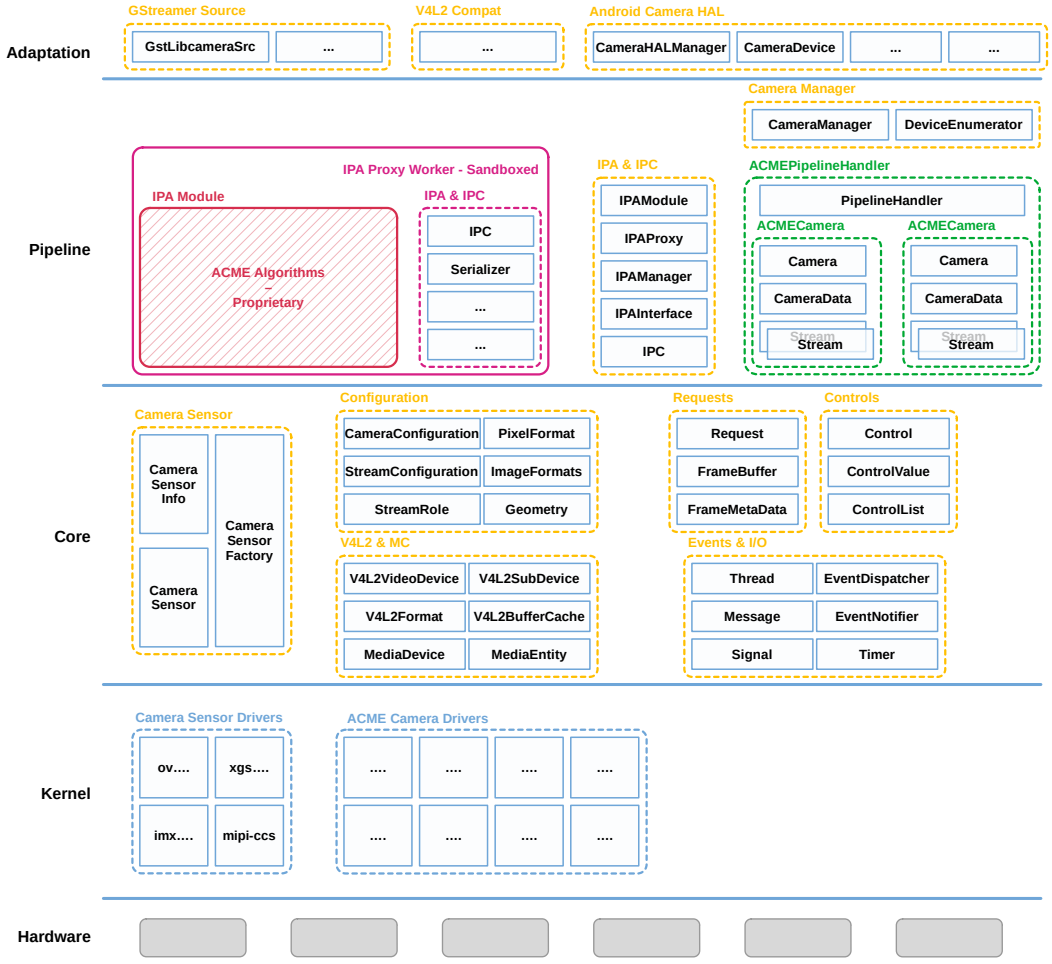
Rockchip ISP

“vimc” (Virtual Media Controller)

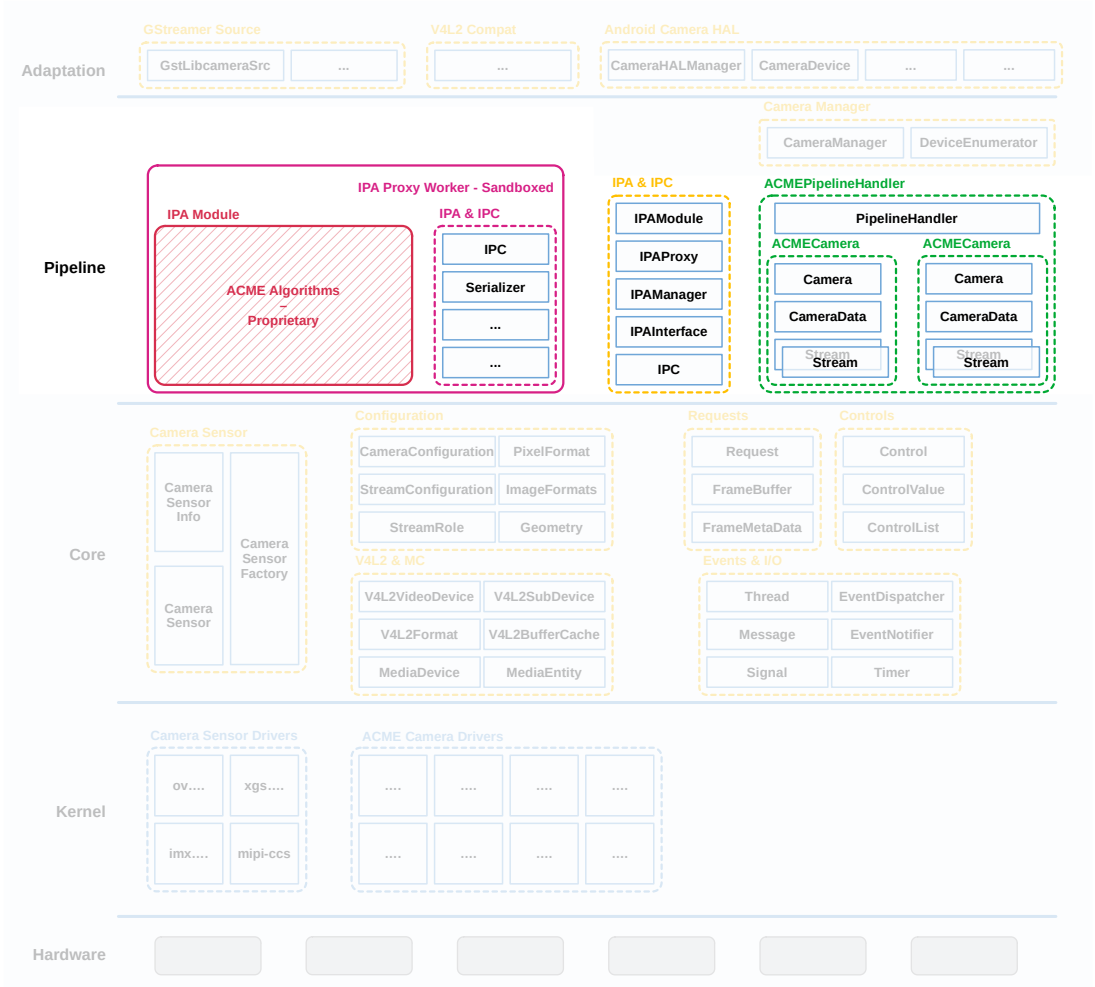
Project Structure – /src/ipa

Controlling The ISP

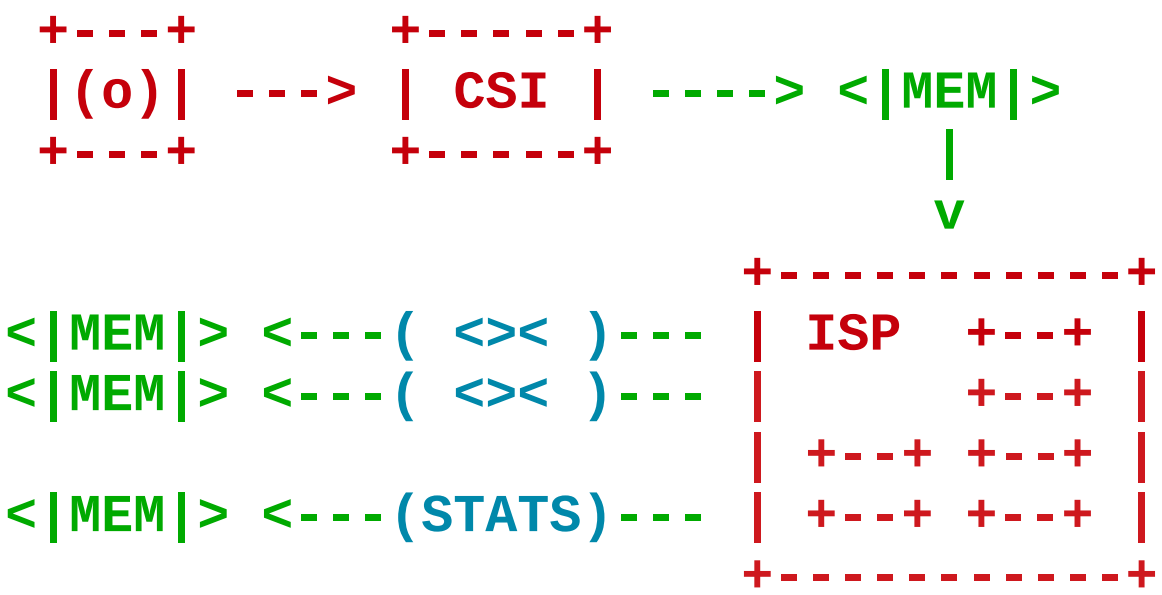
IDEAS ON BOARD



The Camera Stack



The Pipeline



The pipeline handler interfaces with all kernel devices. It abstracts them and exposes video streams to upper layers.

The Pipeline Handler

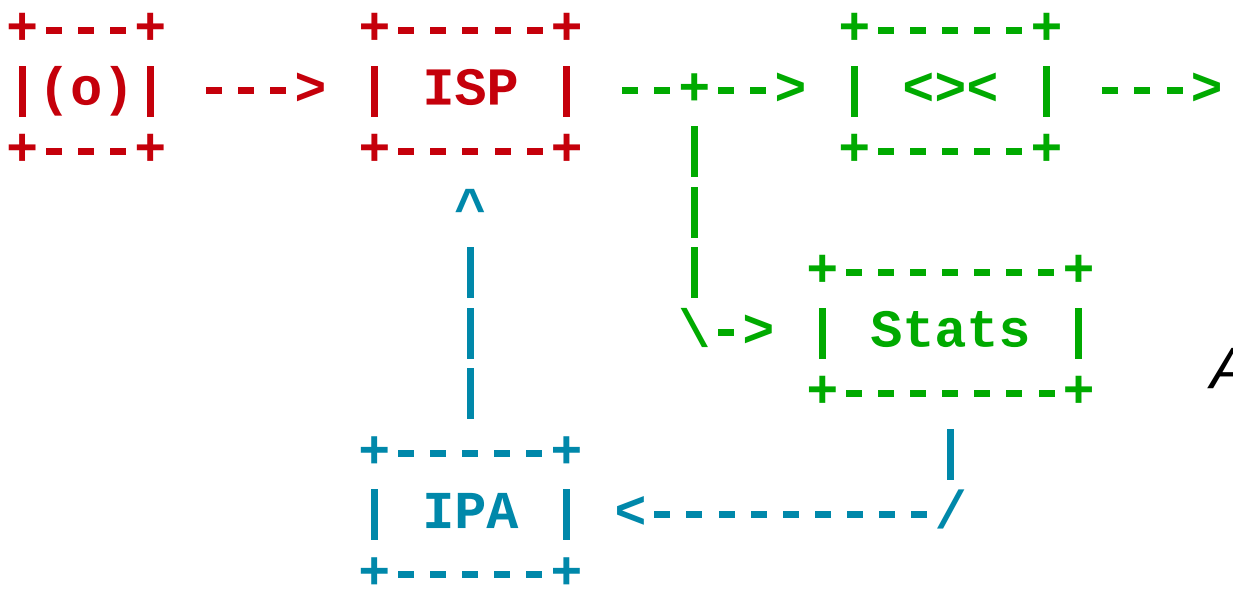
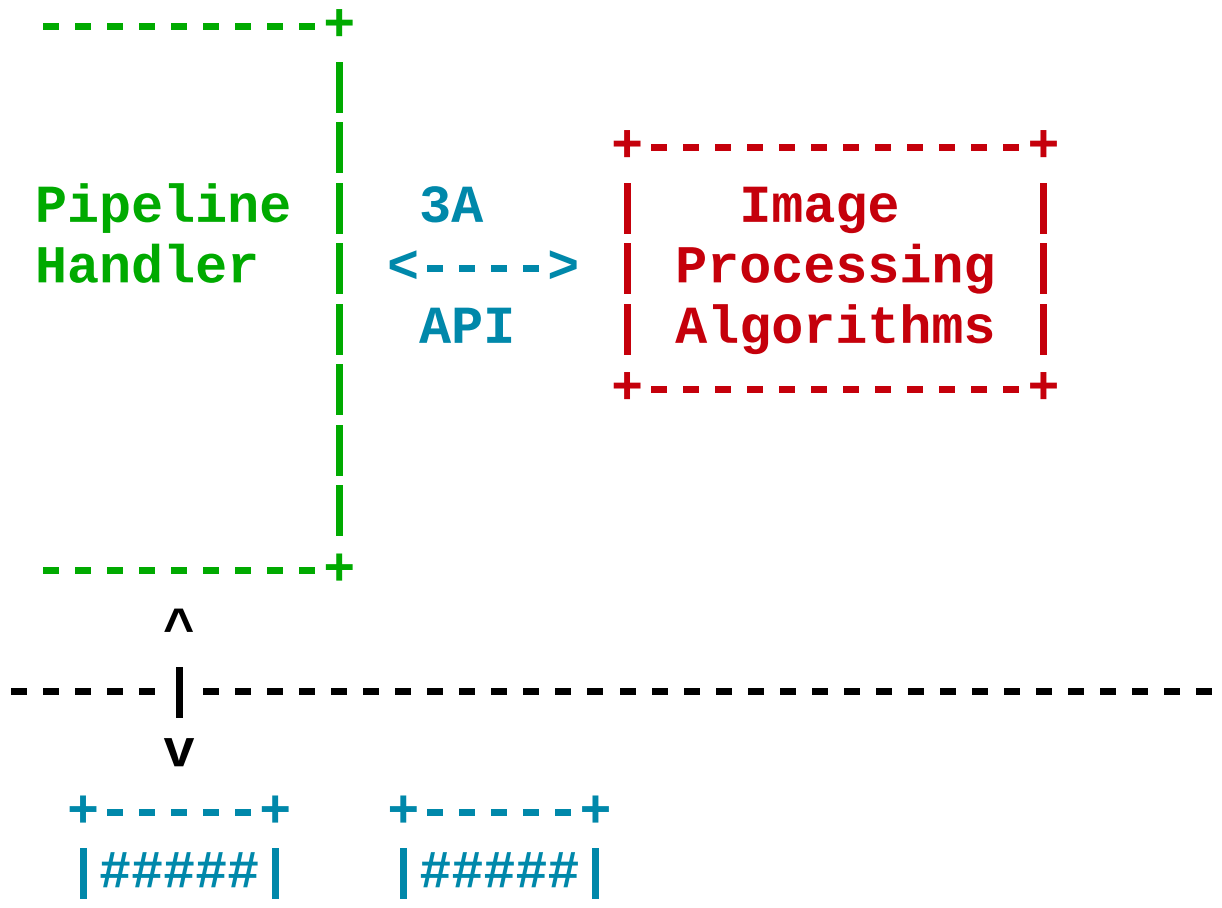


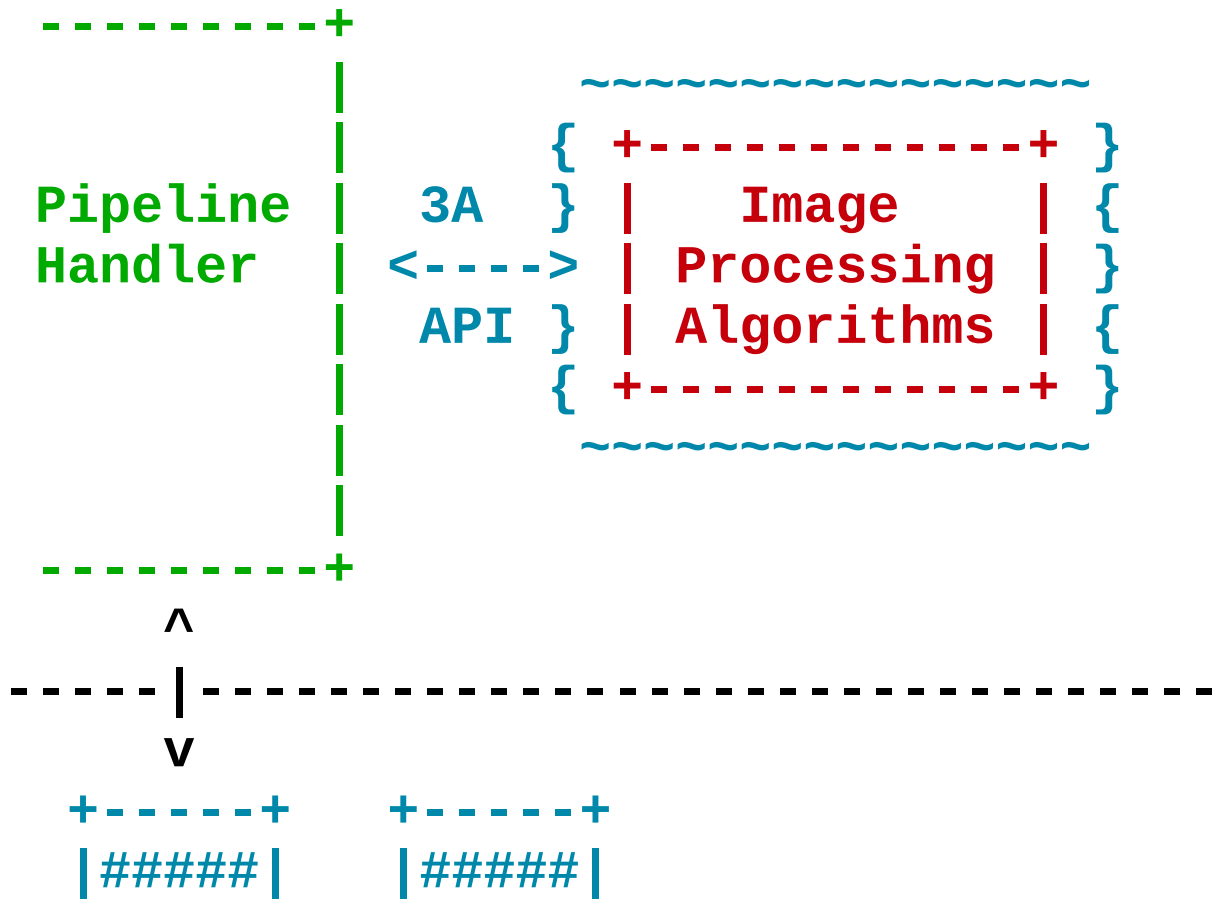
Image Processing Algorithms (IPA) receive statistics from the hardware and compute optimal image parameters.

The Image Processing Algorithms



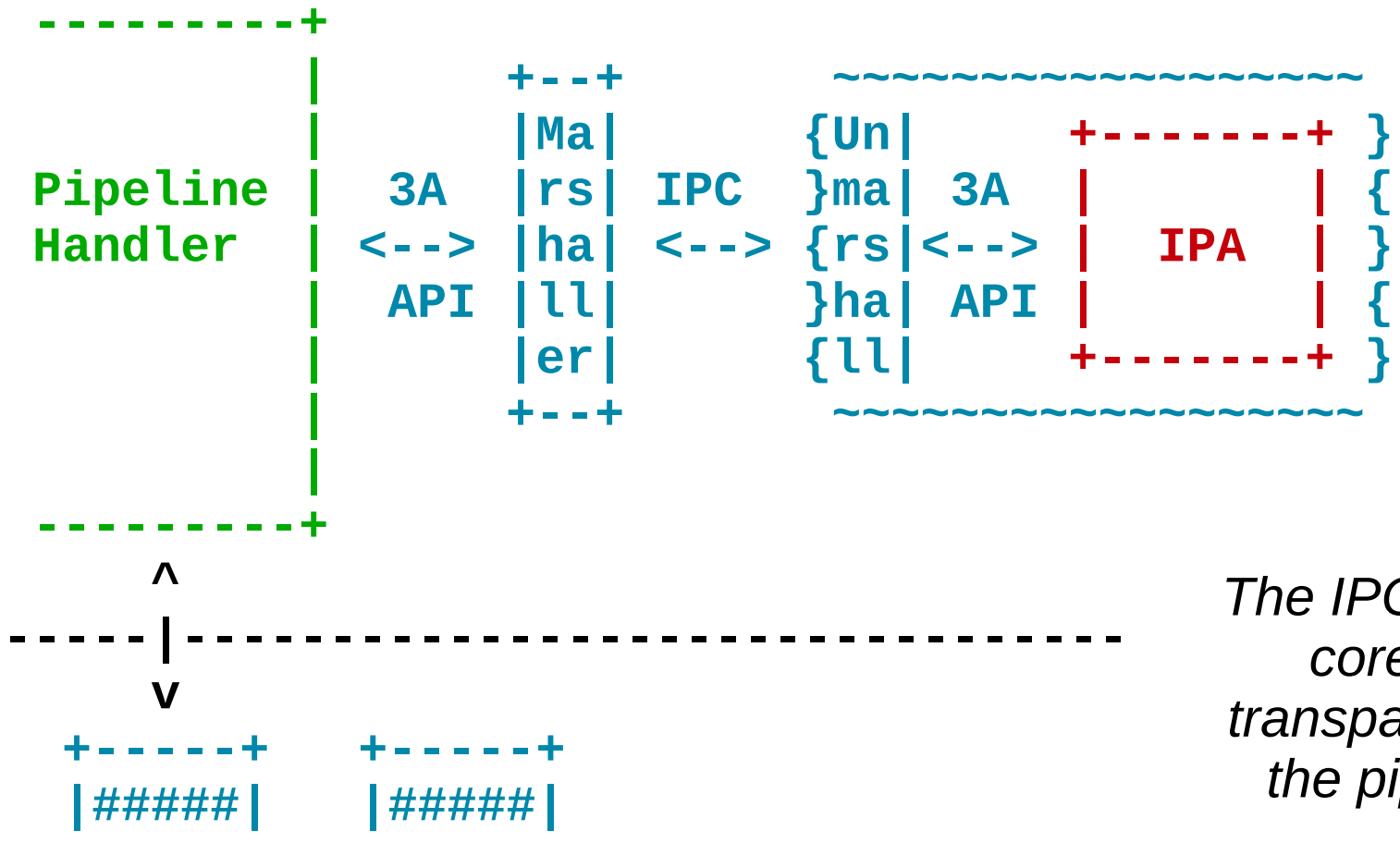
IPAs are separate modules that don't access kernel devices directly. They only have access to their pipeline handler through the IPA API.

The Image Processing Algorithms



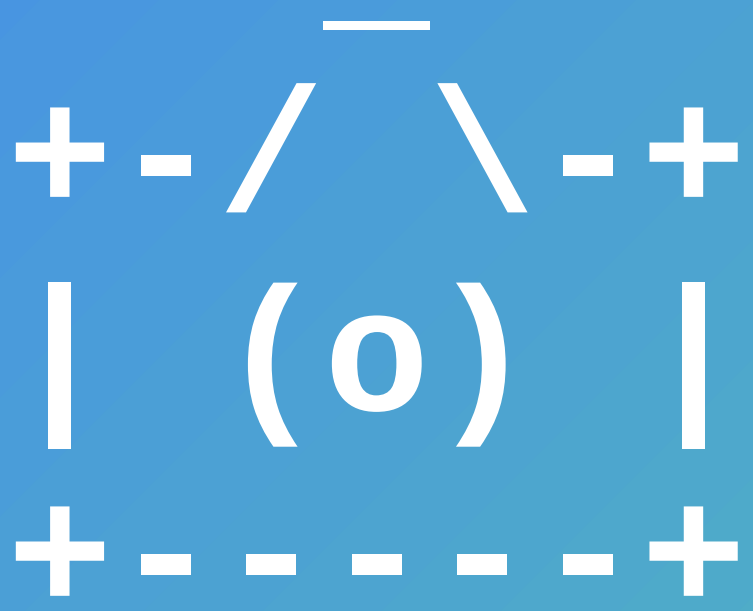
Out-of-tree (including closed-source) IPAs are sandboxed in a separate process. They communicate with the pipeline handler through IPC.

The Image Processing Algorithms



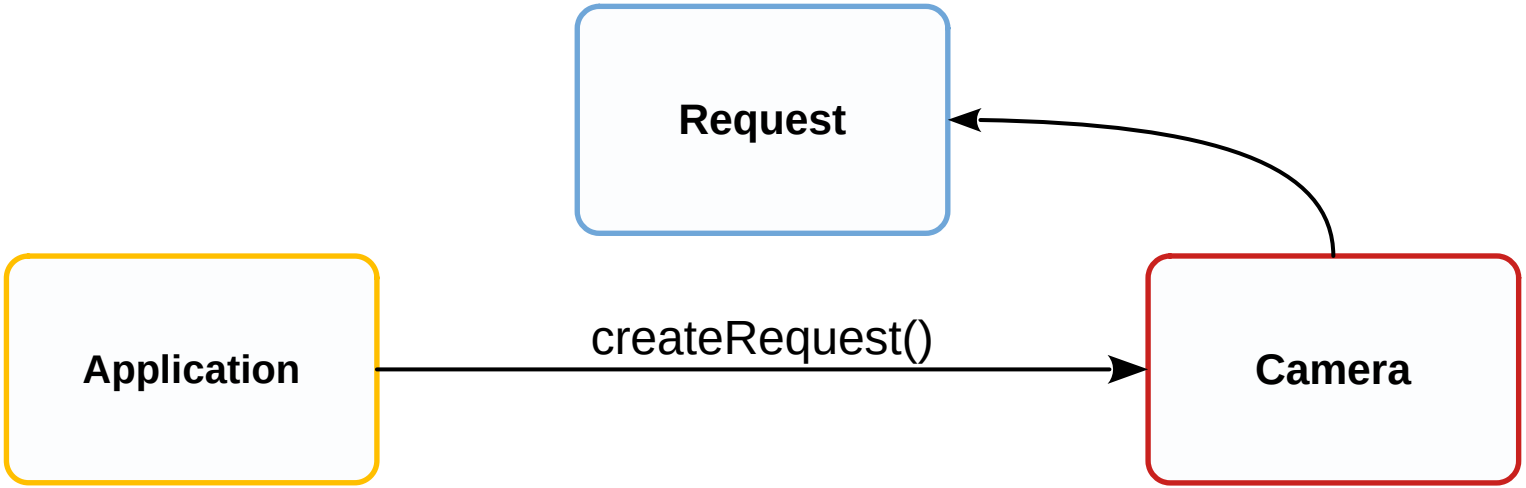
The IPC is handled in core components, transparently for both the pipeline handler and the IPA.

The Image Processing Algorithms

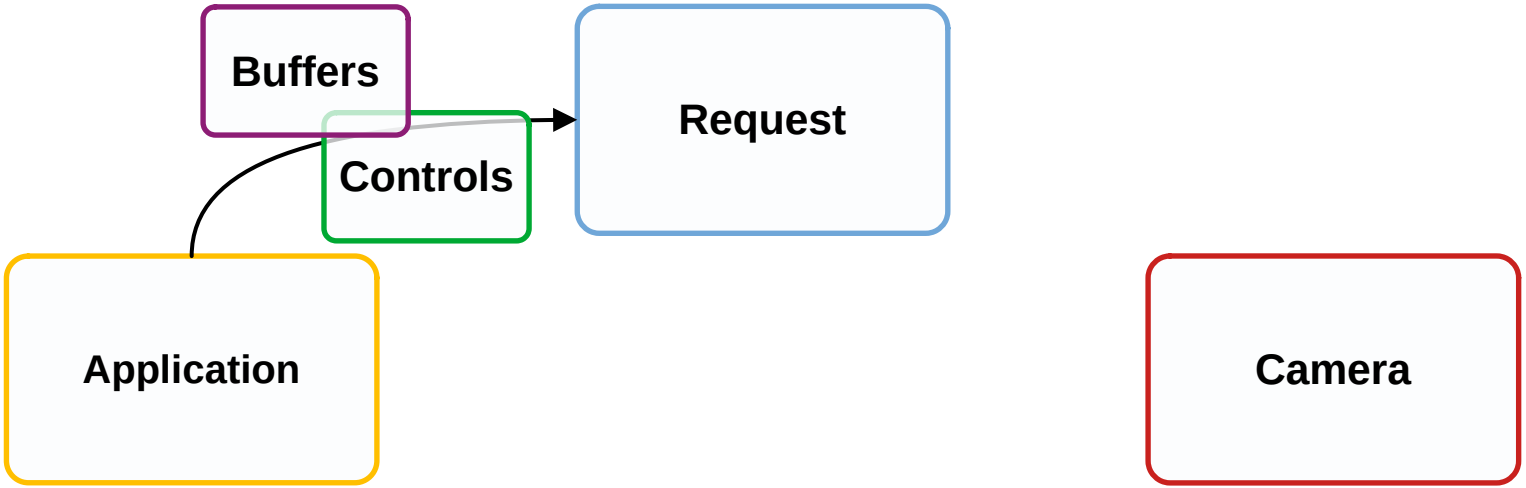


Requests

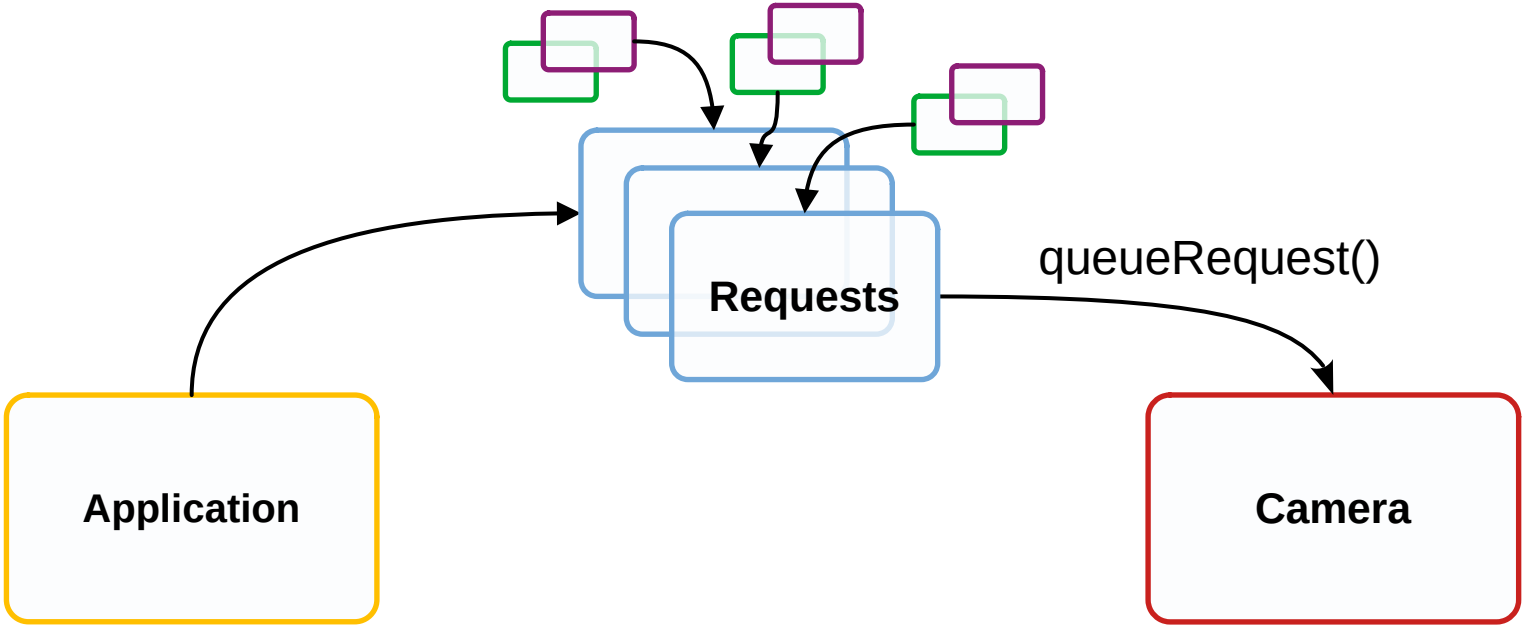
IDEAS ON BOARD



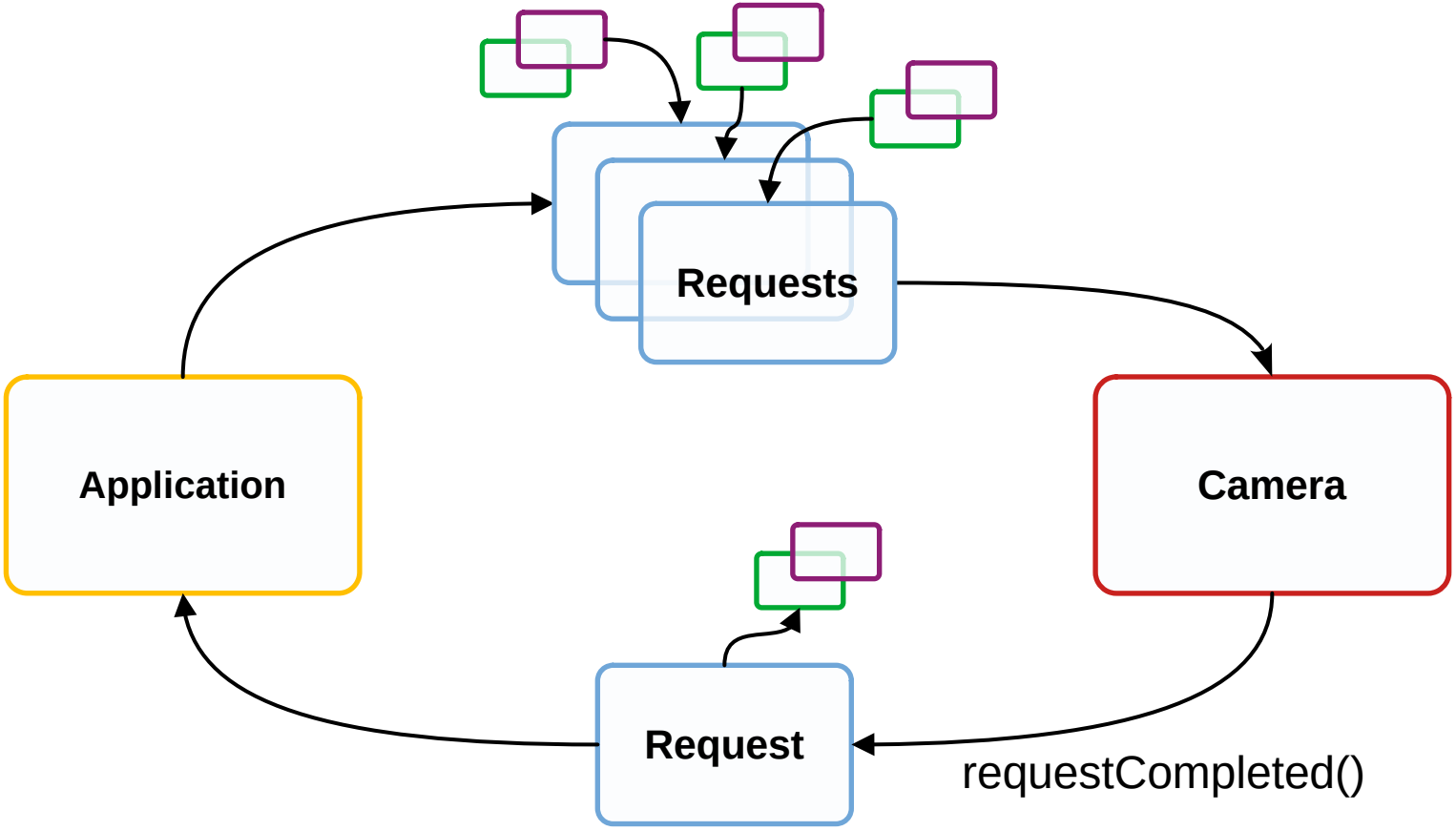
Capture Requests



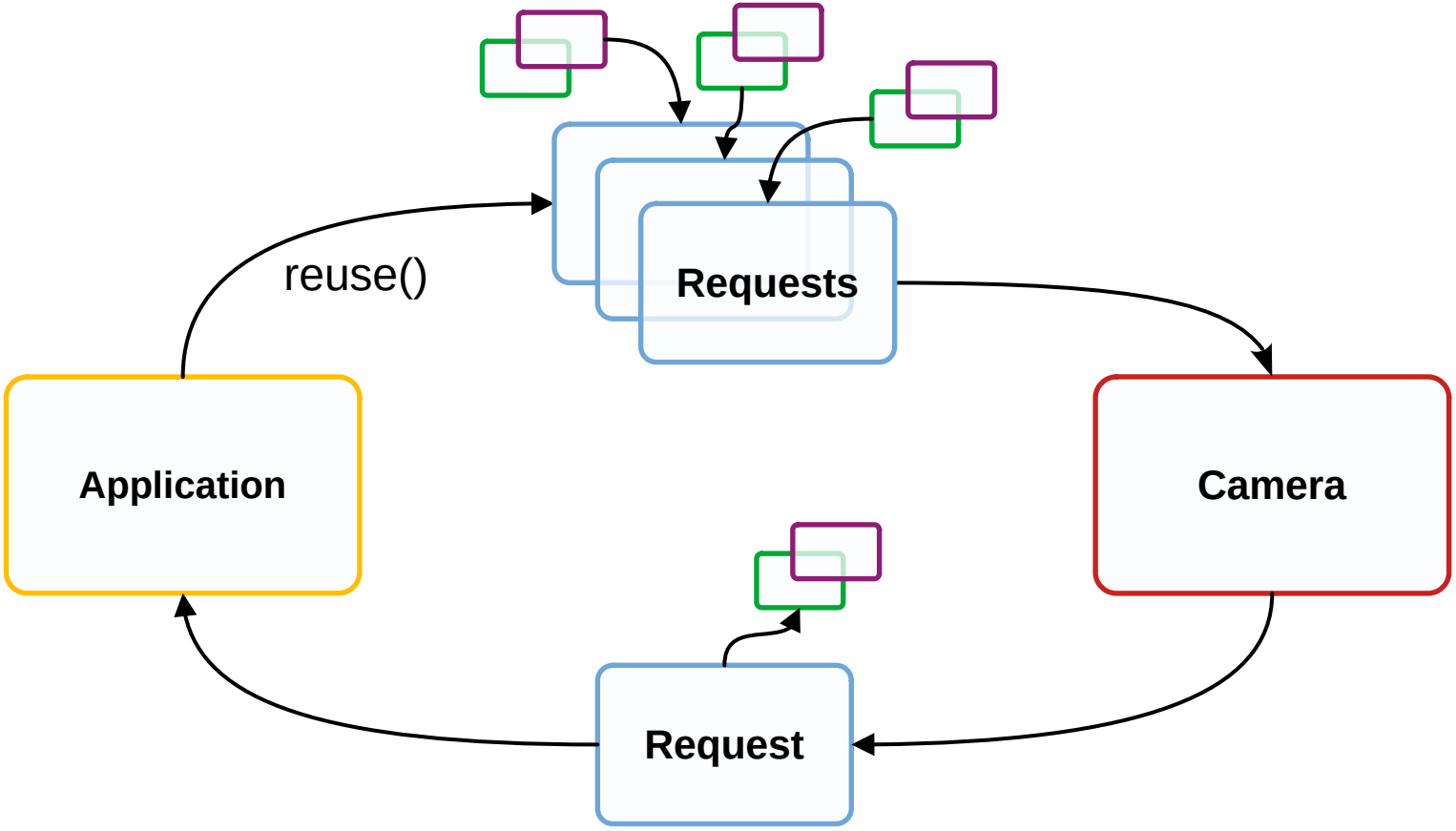
Capture Requests



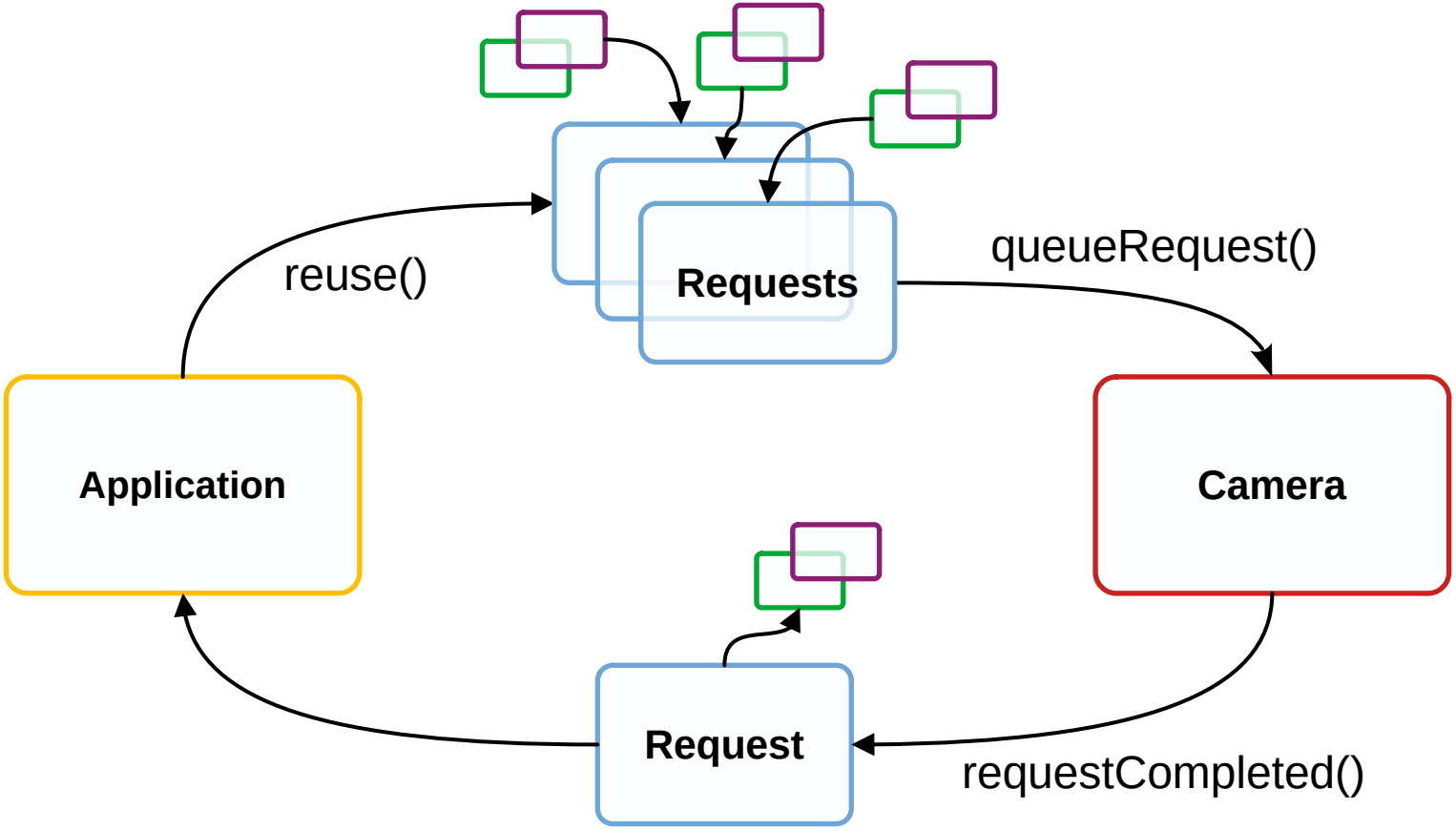
Capture Requests



Capture Requests



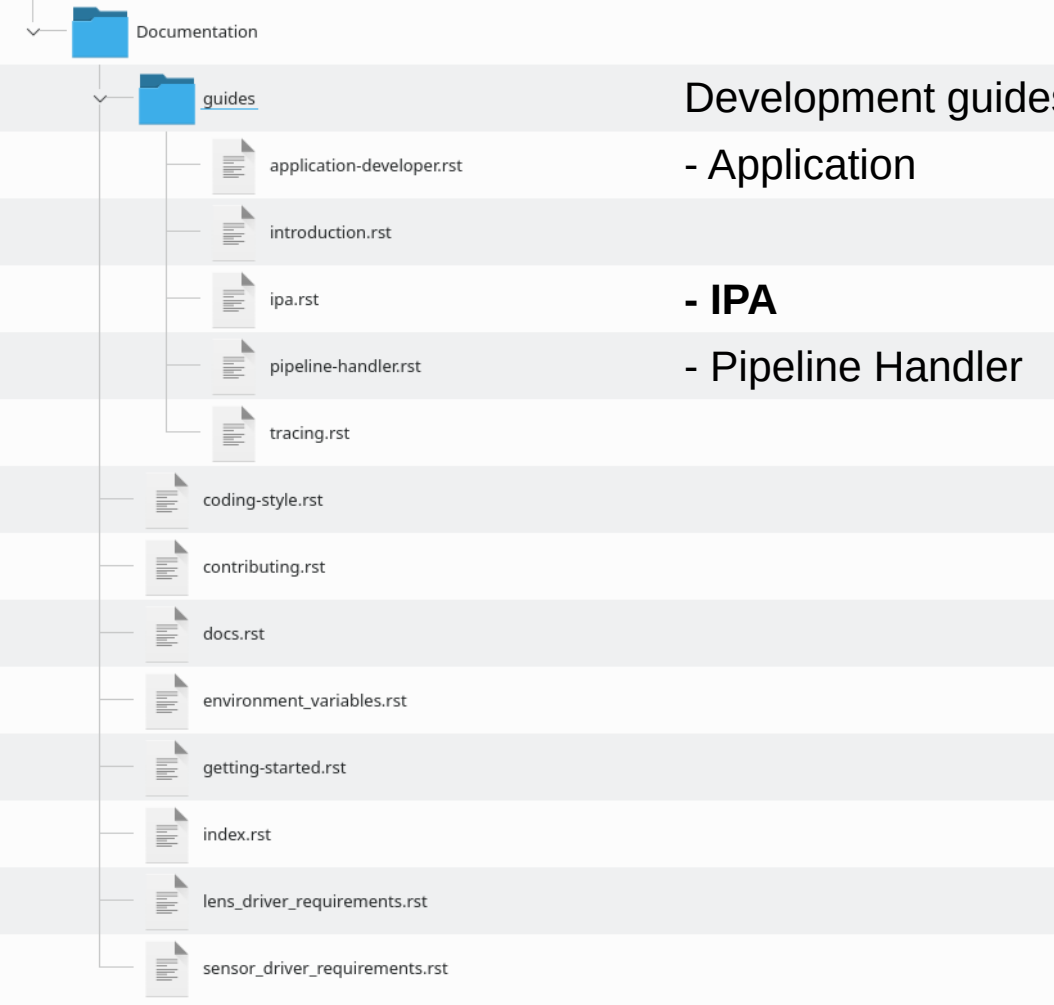
Capture Requests



Capture Requests

IPA Modules

IDEAS ON BOARD



IPA Writer’s Guide

IPA Writer’s Guide

IPA modules are Image Processing Algorithm modules. They provide functionality that the pipeline handler can use for image processing.

This guide covers the definition of the IPA interface, and how to plumb the connection between the pipeline handler and the IPA.

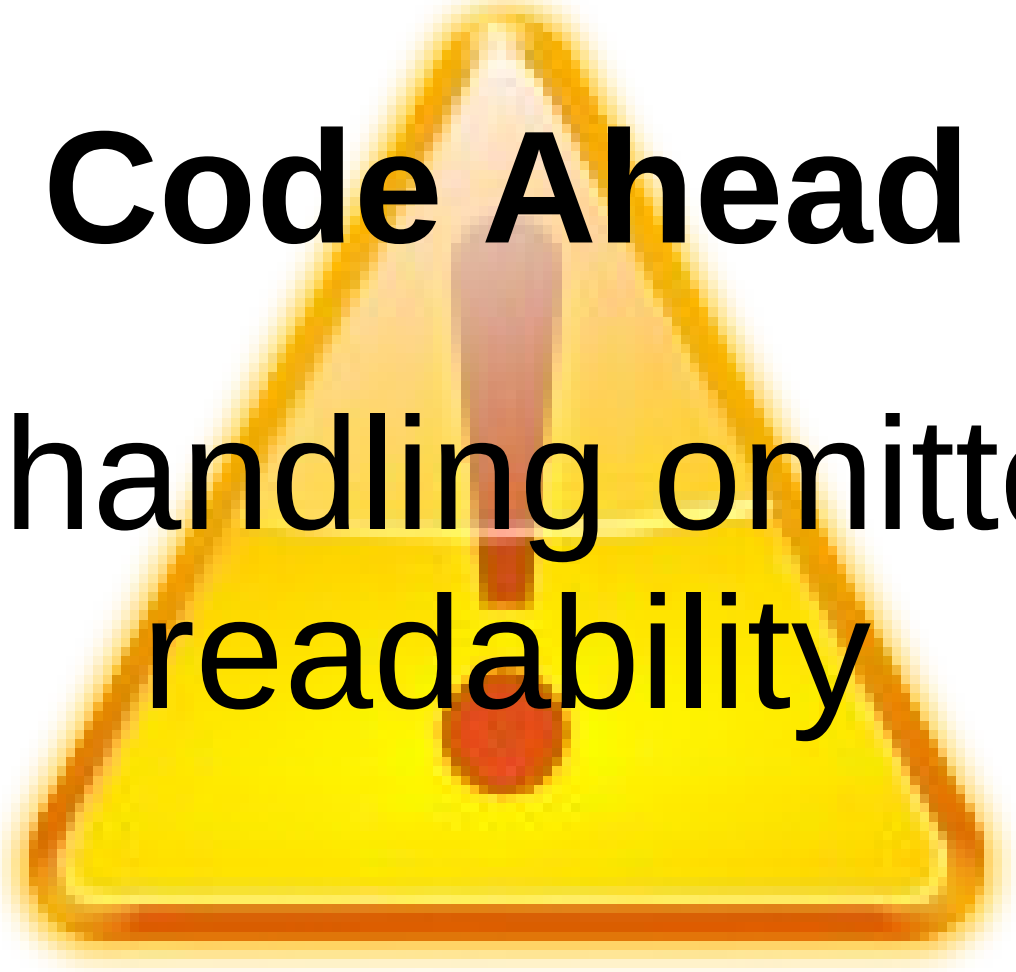
The IPA interface and protocol

The IPA interface defines the interface between the pipeline handler and the IPA. Specifically, it defines the functions that the IPA exposes that the pipeline handler can call, and the signals that the pipeline handler can connect to, in order to receive data from the IPA asynchronously. In addition, it contains any custom data structures that the pipeline handler and IPA may pass to each other.

It is possible to use the same IPA interface with multiple pipeline handlers on different hardware platforms. Generally in such cases, these platforms would have a common hardware

IPA Writer’s Guide

Code Ahead



Error handling omitted for readability

Don’t focus too much on the code.

Don’t focus too much on the code.

It’s C++.

Don’t focus too much on the code.

It’s C++.

With templates.

Don’t focus too much on the code.

It’s C++.

With templates.

And it doesn’t event compile.

1. The IPA Interface
2. The Pipeline Handler
3. The IPA Module
4. The Algorithms

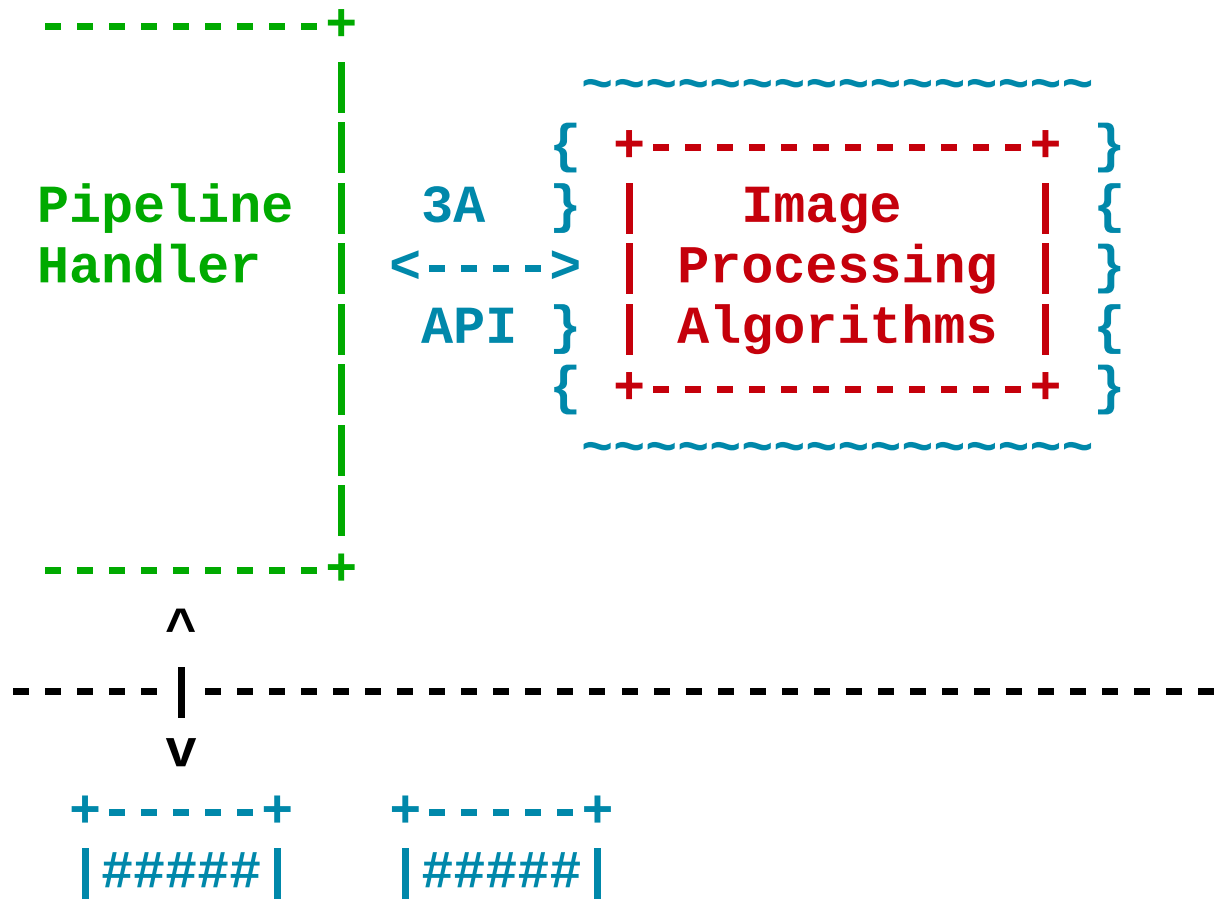
The IPA Interface

IDEAS ON BOARD

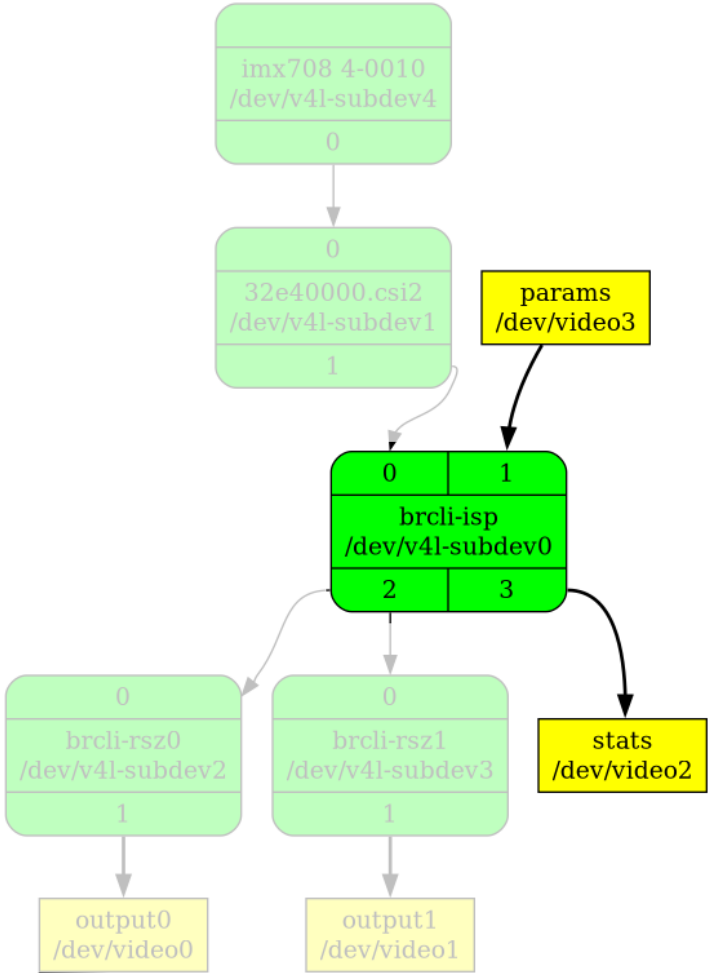
1. The IPA Interface
2. The Pipeline Handler
3. The IPA Module
4. The Algorithms

- Define the interface between the Pipeline Handler and IPA Module

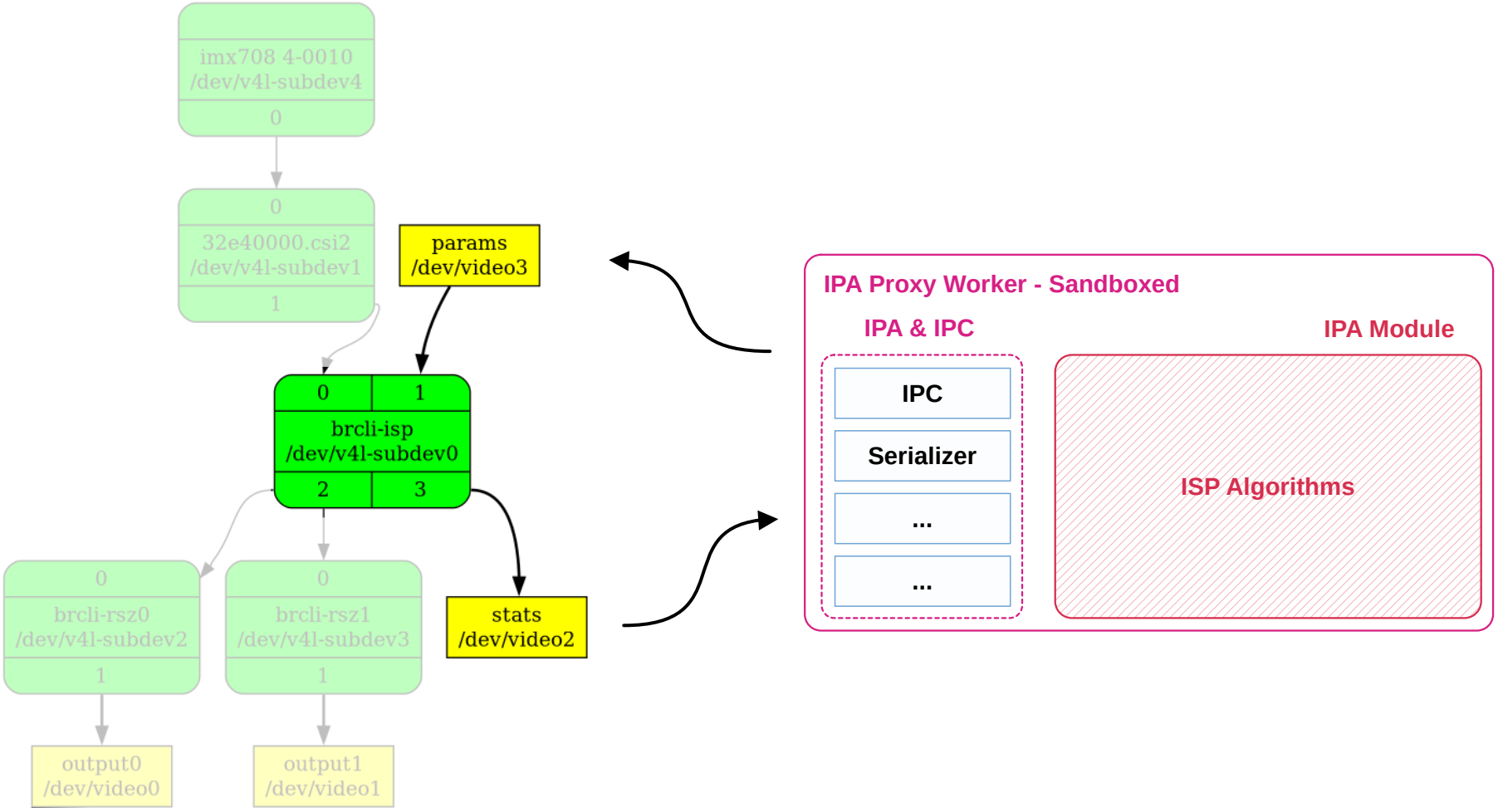
The IPA Interface



The IPA Interface



The IPA Interface



The IPA Interface

```
module ipa.brcli;

import "include/libcamera/ipa/core.mojom";

interface IPABrcliInterface {
     init(libcamera.IPASettings settings,
          uint32 hwRevision,
          libcamera.IPACameraSensorInfo sensorInfo,
          libcamera.ControlInfoMap sensorControls)
          => (int32 ret, libcamera.ControlInfoMap ipaControls);

     configure(IPAConfigInfo configInfo,
            map<uint32, libcamera.IPAStream> streamConfig)
          => (int32 ret, libcamera.ControlInfoMap ipaControls);

     mapBuffers(array<libcamera.IPABuffer> buffers);
     unmapBuffers(array<uint32> ids);

     start() => (int32 ret);
     stop();

     [async] queueRequest(uint32 frame, libcamera.ControlList reqControls);
     [async] processStatsBuffer(uint32 frame, uint32 bufferId,
                        libcamera.ControlList sensorControls);
     [async] fillParamsBuffer(uint32 frame, uint32 bufferId);
};
```

The IPA interface defines the operations exposed by the IPA module to the pipeline handler. It is expressed using mojom IDL.

include/libcamera/ipa/brcli.mojom

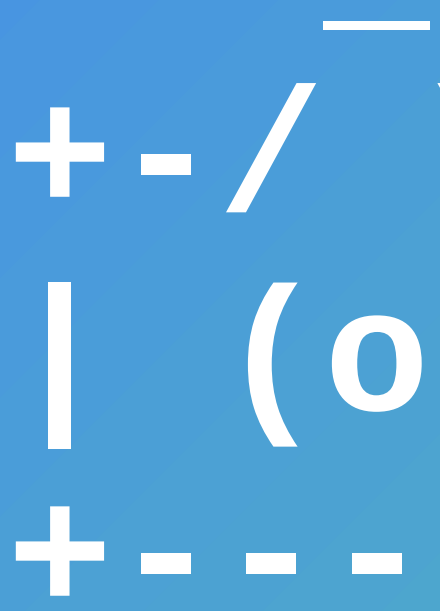
The IPA Interface

```
module ipa.brcli;

import "include/libcamera/ipa/core.mojom";

...

interface IPABrcliEventInterface {
     paramsBufferReady(uint32 frame);
     setSensorControls(uint32 frame, libcamera.ControlList sensorControls);
     metadataReady(uint32 frame, libcamera.ControlList metadata);
};
```

The IPA event interface defines the events generated by the IPA module for the pipeline handler.

include/libcamera/ipa/brcli.mojom

The IPA Interface



The Pipeline Handler

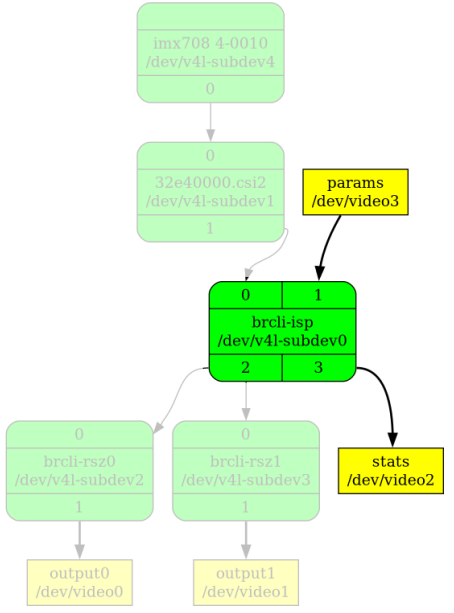
IDEAS ON BOARD

1. The IPA Interface
2. The Pipeline Handler
3. The IPA Module
4. The Algorithms

- Communicate with the IPA module
- Wire up the statistics capture and ISP parameters

The Pipeline Handler

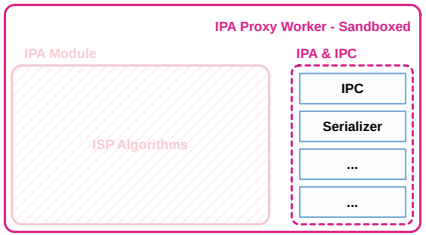
```
bool PipelineHandlerBrcli::match(DeviceEnumerator *enumerator)
{
...
        ipa_ = IPAManager::createIPA<ipa::brcli::IPAProxyBrcli1>(pipe(), 1, 1);
        if (!ipa_)
                return -ENOENT;
...
}
```



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Loading the IPA

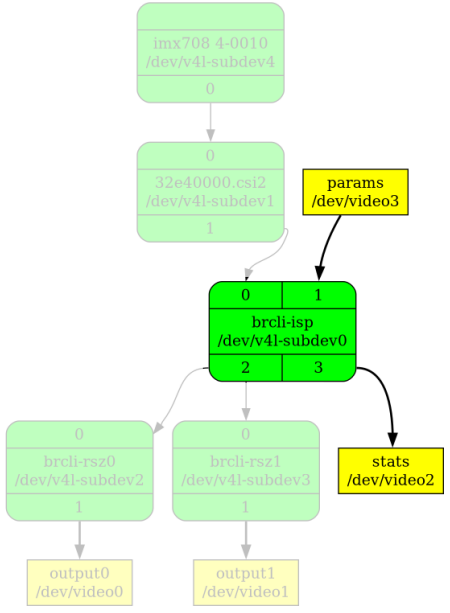
```
module ipa.brcli;

import "include/libcamera/ipa/core.mojom";

...

interface IPABrcliEventInterface {
     paramsBufferReady(uint32 frame);
     setSensorControls(uint32 frame, libcamera.ControlList sensorControls);
     metadataReady(uint32 frame, libcamera.ControlList metadata);
};
```



include/libcamera/ipa/brcli.mojom

Pipeline Handler – Loading the IPA

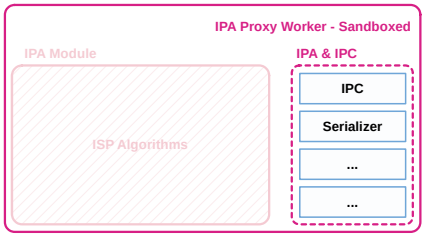
```
bool PipelineHandlerBrcli::match(DeviceEnumerator *enumerator)
{
...
        ipa_->paramsBufferReady.connect(this, &BrcliCameraData::paramFilled);
        ipa_->setSensorControls.connect(this, &BrcliCameraData::setSensorControls);
        ipa_->metadataReady.connect(this, &BrcliCameraData::metadataReady);
...
}
```



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Loading the IPA

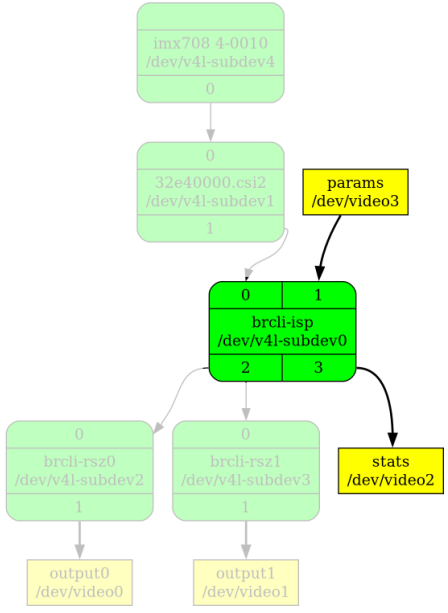
```
module ipa.brcli;

import "include/libcamera/ipa/core.mojom";

interface IPABrcliInterface {
     init(libcamera.IPASettings settings,
          uint32 hwRevision,
          libcamera.IPACameraSensorInfo sensorInfo,
          libcamera.ControlInfoMap sensorControls)
          => (int32 ret, libcamera.ControlInfoMap ipaControls);

     configure(IPAConfigInfo configInfo,
               map<uint32, libcamera.IPAStream> streamConfig)
          => (int32 ret, libcamera.ControlInfoMap ipaControls);

     mapBuffers(array<libcamera.IPABuffer> buffers);
     unmapBuffers(array<uint32> ids);

     start() => (int32 ret);
     stop();

     [async] queueRequest(uint32 frame, libcamera.ControlList reqControls);
     [async] processStatsBuffer(uint32 frame, uint32 bufferId,
                                libcamera.ControlList sensorControls);
     [async] fillParamsBuffer(uint32 frame, uint32 bufferId);
};
```



include/libcamera/ipa/brcli.mojom

Pipeline Handler – Loading the IPA

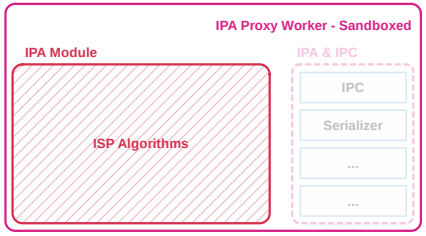
```
bool PipelineHandlerBrcli::match(DeviceEnumerator *enumerator)
{
...
        std::string ipaTuningFile;
        char const *configFromEnv = utils::secure_getenv("LIBCAMERA_BRCLI_TUNING_FILE");
        if (!configFromEnv || *configFromEnv == '\0') {
                ipaTuningFile = ipa_->configurationFile(sensor_->model() + ".yaml");
                /*
                 * If the tuning file isn't found, fall back to the
                 * 'uncalibrated' configuration file.
                 */
                if (ipaTuningFile.empty())
                        ipaTuningFile = ipa_->configurationFile("uncalibrated.yaml"
        } else {
                ipaTuningFile = std::string(configFromEnv);
        }

        IPACameraSensorInfo sensorInfo{};
        sensor_->sensorInfo(&sensorInfo);

        ipa_->init({ ipaTuningFile, sensor_->model() }, hwRevision,
                   sensorInfo, sensor_->controls(), &controlInfo_);
...
}
```



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Loading the IPA

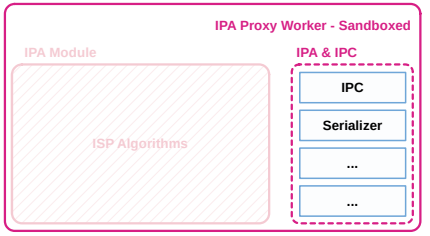
```yaml
# SPDX-License-Identifier: CC0-1.0
%YAML 1.1
---
version: 1
algorithms:
  - Agc:
  - Awb:
  - BlackLevelCorrection:
      R:  256
      Gr: 256
      Gb: 256
      B:  256
...
```



src/ipa/brcli/data/imx219.yaml

Tuning File

```
module ipa.brcli;

import "include/libcamera/ipa/core.mojom";

interface IPABrcliInterface {
    init(libcamera.IPASettings settings,
         uint32 hwRevision,
         libcamera.IPACameraSensorInfo sensorInfo,
         libcamera.ControlInfoMap sensorControls)
         => (int32 ret, libcamera.ControlInfoMap ipaControls);

    configure(IPAConfigInfo configInfo,
              map<uint32, libcamera.IPAStream> streamConfig)
         => (int32 ret, libcamera.ControlInfoMap ipaControls);

    mapBuffers(array<libcamera.IPABuffer> buffers);
    unmapBuffers(array<uint32> ids);

    start() => (int32 ret);
    stop();

    [async] queueRequest(uint32 frame, libcamera.ControlList reqControls);
    [async] processStatsBuffer(uint32 frame, uint32 bufferId,
                               libcamera.ControlList sensorControls);
    [async] fillParamsBuffer(uint32 frame, uint32 bufferId);
};
```

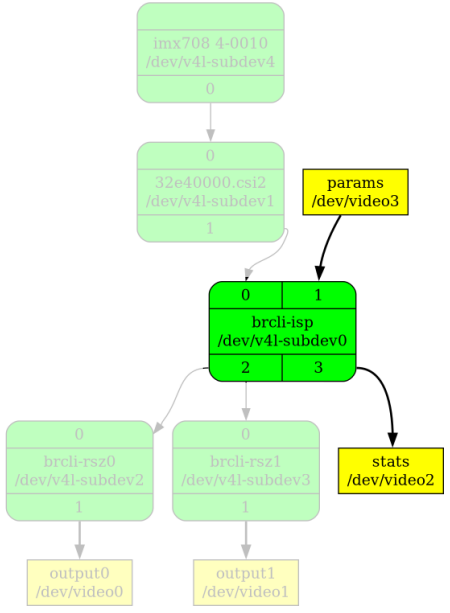


include/libcamera/ipa/brcli.mojom

IDEAS ON BOARD

Pipeline Handler – Configuring the IPA

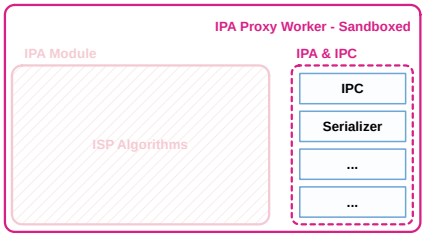
```
int PipelineHandlerBrcli::configure(Camera *camera, CameraConfiguration *c)
{
        BrcliCameraConfiguration *config =
                static_cast<BrcliCameraConfiguration *>(c);
        BrcliCameraData *data = cameraData(camera);
...
        std::map<unsigned int, IPAStream> streamConfig;

        for (const StreamConfiguration &cfg : *config) {
                if (cfg.stream() == &data->mainPathStream_)
                        streamConfig[0] = IPAStream(cfg.pixelFormat,
                                                    cfg.size);
                else
                        streamConfig[1] = IPAStream(cfg.pixelFormat,
                                                    cfg.size);
        }

        data->ipa_->configure(ipaConfig, streamConfig, &data->controlInfo_);
...
}
```



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Configuring the IPA

```
module ipa.brcli;

import "include/libcamera/ipa/core.mojom";

interface IPABrcliInterface {
    init(libcamera.IPASettings settings,
         uint32 hwRevision,
         libcamera.IPACameraSensorInfo sensorInfo,
         libcamera.ControlInfoMap sensorControls)
         => (int32 ret, libcamera.ControlInfoMap ipaControls);

    configure(IPAConfigInfo configInfo,
           map<uint32, libcamera.IPAStream> streamConfig)
         => (int32 ret, libcamera.ControlInfoMap ipaControls);

    mapBuffers(array<libcamera.IPABuffer> buffers);
    unmapBuffers(array<uint32> ids);

    start(libcamera.ControlList controls) => (int32 ret);
    stop();

    [async] queueRequest(uint32 frame, libcamera.ControlList reqControls);
    [async] processStatsBuffer(uint32 frame, uint32 bufferId,
                               libcamera.ControlList sensorControls);
    [async] fillParamsBuffer(uint32 frame, uint32 bufferId);
};
```

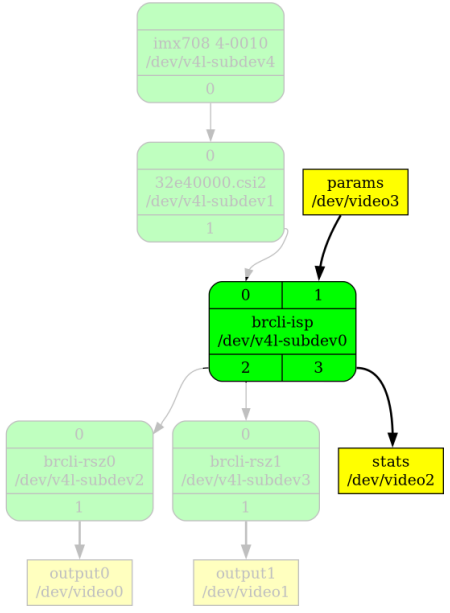


include/libcamera/ipa/brcli.mojom

Pipeline Handler – Starting the IPA

```
int PipelineHandlerBrcli::start(Camera *camera, const ControlList *controls)
{
        BrcliCameraData *data = cameraData(camera);
...
        ret = data->ipa_->start(controls);
        if (ret) {
                LOG(Brcli, Error) << "Failed to start IPA";
                return ret;
        }
...
}
```

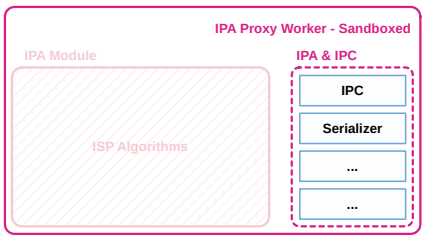
The IPA module now runs, in a separate thread or process.



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Starting the IPA

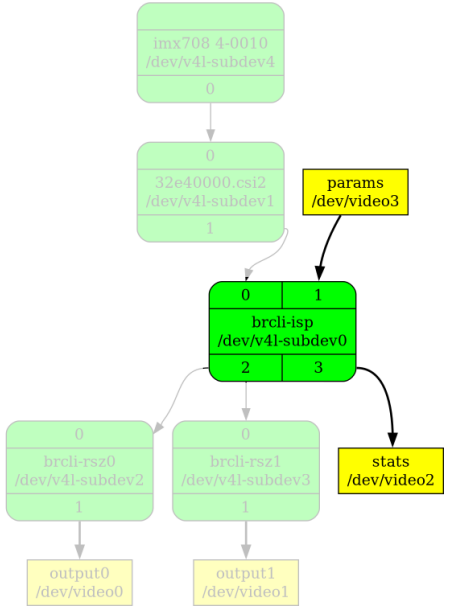
```
module ipa.brcli;

import "include/libcamera/ipa/core.mojom";

interface IPABrcliInterface {
    init(libcamera.IPASettings settings,
         uint32 hwRevision,
         libcamera.IPACameraSensorInfo sensorInfo,
         libcamera.ControlInfoMap sensorControls)
        => (int32 ret, libcamera.ControlInfoMap ipaControls);

    configure(IPAConfigInfo configInfo,
          map<uint32, libcamera.IPAStream> streamConfig)
        => (int32 ret, libcamera.ControlInfoMap ipaControls);

    mapBuffers(array<libcamera.IPABuffer> buffers);
    unmapBuffers(array<uint32> ids);

    start(libcamera.ControlList controls) => (int32 ret);
    stop();

    [async] queueRequest(uint32 frame, libcamera.ControlList reqControls);
    [async] processStatsBuffer(uint32 frame, uint32 bufferId,
                               libcamera.ControlList sensorControls);
    [async] fillParamsBuffer(uint32 frame, uint32 bufferId);
};
```



include/libcamera/ipa/brcli.mojom

Pipeline Handler – Runtime

```
int PipelineHandlerBrcli::queueRequestDevice(Camera *camera, Request *request)
{
        BrcliCameraData *data = cameraData(camera);
...
        BrcliFrameInfo *info = data->frameInfo_.create(data, request);
        if (!info)
                return -ENOENT;

        data->ipa_->queueRequest(data->frame_, request->controls());

        data->frame_++;
...
}
```

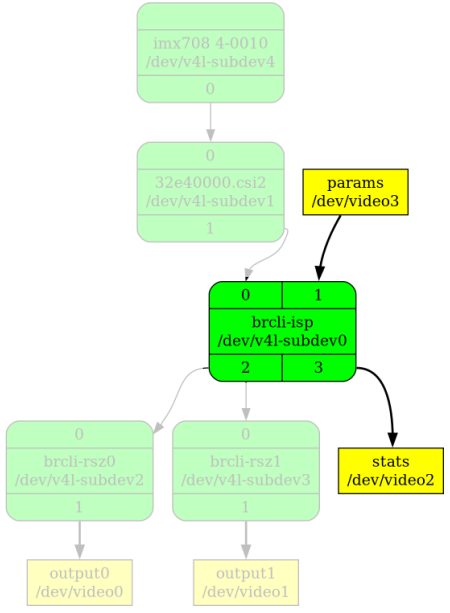


src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Queuing a Request

```
int PipelineHandlerBrcli::queueRequestDevice(Camera *camera, Request *request)
{
        BrcliCameraData *data = cameraData(camera);
...
        BrcliFrameInfo *info = data->frameInfo_.create(data, request);
        if (!info)
                return -ENOENT;

        data->ipa_->queueRequest(data->frame_, request->controls());

        data->frame_++;
...
}
```

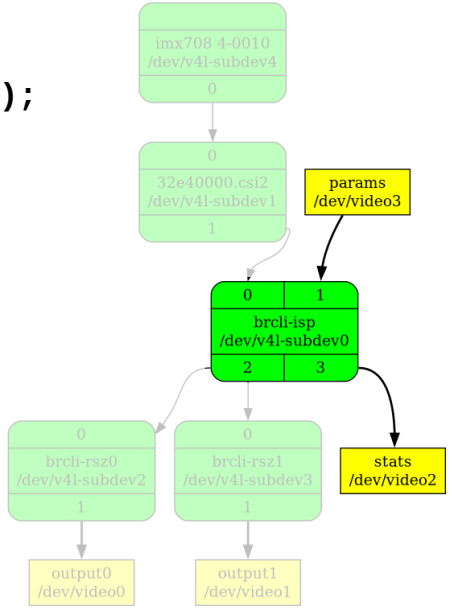
Everything at runtime is asynchronous.



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Queuing a Request

```
int PipelineHandlerBrcli::statsReady(FrameBuffer *buffer)
{
        BrcliCameraData *data = cameraData(camera);
...
        BrcliFrameInfo *info = data->frameInfo_.find(buffer);
        if (!info)
                return;

        data->ipa_->processStatsBuffer(info->frame, info->statBuffer->cookie(),
                                       data->delayedCtrls_->get(buffer->metadata().sequence);
...
}
```

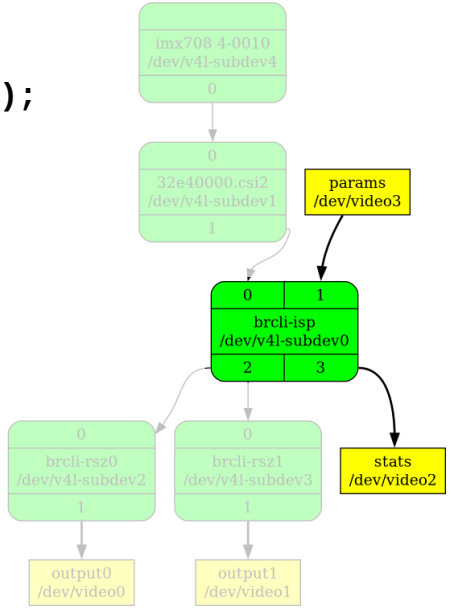


src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Statistics Ready

```
int PipelineHandlerBrcli::statsReady(FrameBuffer *buffer)
{
        BrcliCameraData *data = cameraData(camera);
...
        BrcliFrameInfo *info = data->frameInfo_.find(buffer);
        if (!info)
                return;

        data->ipa_->processStatsBuffer(info->frame, info->statBuffer->cookie(),
                                       data->delayedCtrls_->get(buffer->metadata().sequence);
...
}
```

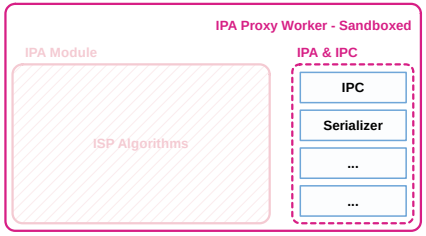
The IPA needs to know what parameters have been used by the sensor for this frame.



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Statistics Ready

```
module ipa.brcli;

import "include/libcamera/ipa/core.mojom";

...

interface IPABrcliEventInterface {
     paramsBufferReady(uint32 frame);
     setSensorControls(uint32 frame, libcamera.ControlList sensorControls);
     metadataReady(uint32 frame, libcamera.ControlList metadata);
};
```

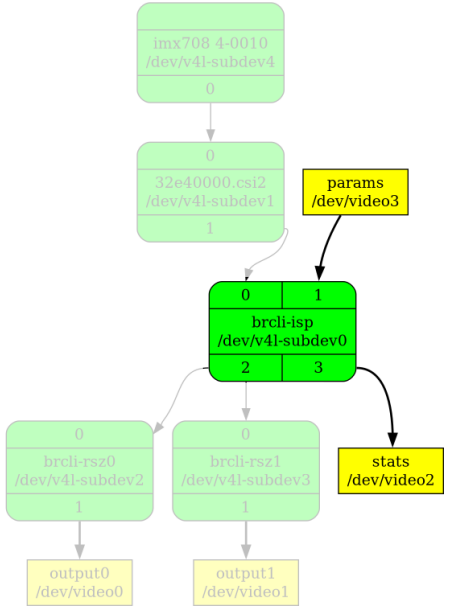


include/libcamera/ipa/brcli.mojom

Pipeline Handler – Events

```
void BrcliCameraData::setSensorControls([[maybe_unused]] unsigned int frame,
                                        const ControlList &sensorControls)
{
        delayedCtrls_->push(sensorControls);
}
```

Sensors typically delay applying controls by a frame or two.



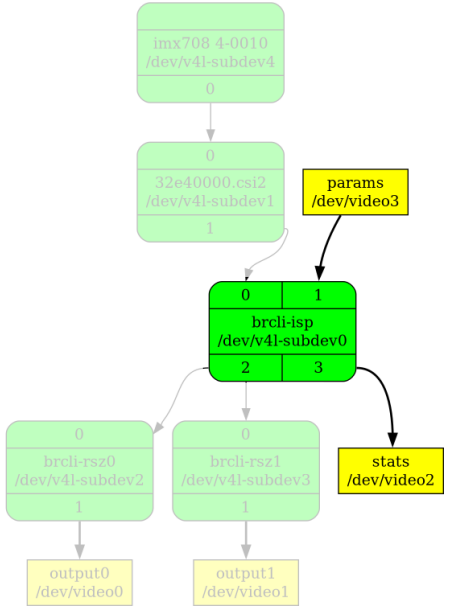
src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Sensor Parameters

```
void BrcliCameraData::setSensorControls([[maybe_unused]] unsigned int frame,
                                        const ControlList &sensorControls)
{
        delayedCtrls_->push(sensorControls);
}
```

Sensors typically delay applying controls by a frame or two.

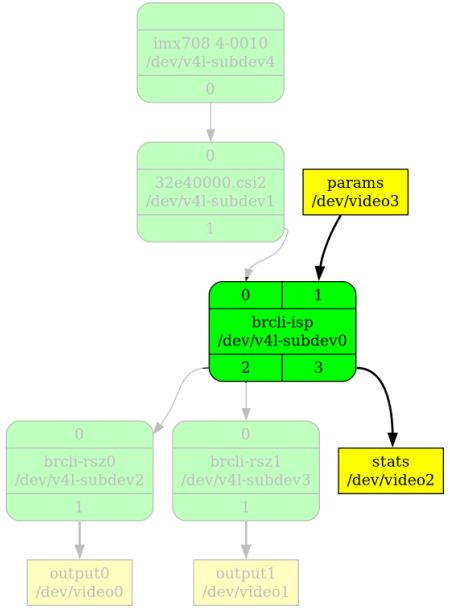
This topic could fill a whole talk by itself.



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Sensor Parameters

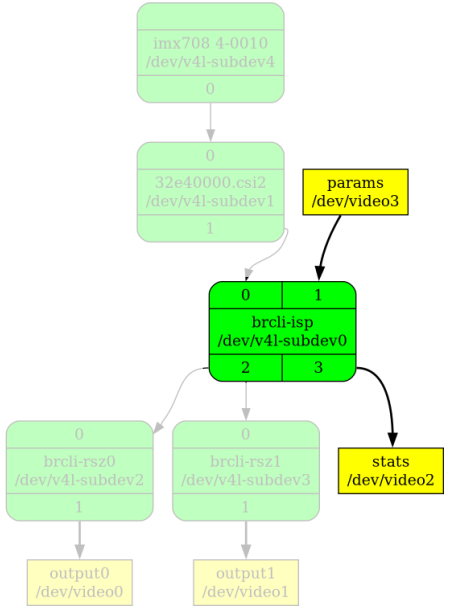
```
int BrcliCameraData::paramFilled(unsigned int frame)
{
        PipelineHandlerBrcli *pipe = BrcliCameraData::pipe();
        BrcliFrameInfo *info = frameInfo_.find(frame);
        if (!info)
                return;

        info->paramBuffer->_d()->metadata().planes()[0].bytesused =
                sizeof(struct brcli_params_cfg);
        pipe->param_->queueBuffer(info->paramBuffer);
...
}
```



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – ISP Parameters

```
int BrcliCameraData::metadataReady(unsigned int frame, const ControlList &metadata)
{
        BrcliFrameInfo *info = frameInfo_.find(frame);
        if (!info)
                return;

        info->request->metadata().merge(metadata);
        info->metadataProcessed = true;

        pipe()->tryCompleteRequest(info);
}
```



src/libcamera/pipeline/brcli/brcli.cpp

Pipeline Handler – Metadata

The IPA Module

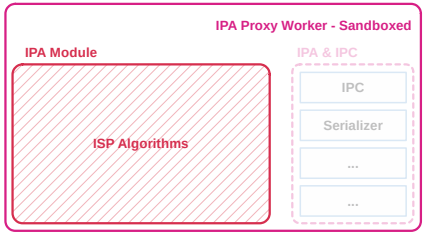
IDEAS ON BOARD

1. The IPA Interface
2. The Pipeline Handler
3. The IPA Module
4. The Algorithms

- Process the statistics and user controls
- Compute the sensor and ISP parameters

The IPA Module

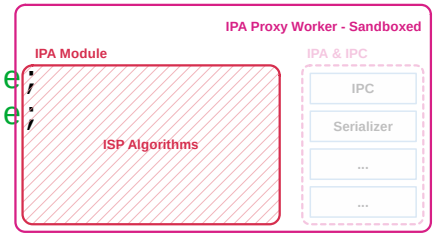
```
/*
 * External IPA module interface
 */

extern "C" {
const struct IPAModuleInfo ipaModuleInfo = {
        IPA_MODULE_API_VERSION,
        1,
        "PipelineHandlerBrcli",
        "brcli",
};

IPAInterface *ipaCreate()
{
        return new ipa::brcli::IPABrcli();
}
}
```



src/ipa/brcli/brcli.cpp

The IPA Module – Entry Point

```
namespace ipa::brcli {

class IPABrcli : public IPABrcliInterface
{
public:
        IPABrcli();

        int init(const IPASettings &settings, unsigned int hwRevision,
                 const IPACameraSensorInfo &sensorInfo,
                 const ControlInfoMap &sensorControls,
                 ControlInfoMap *ipaControls) override;
        int start() override;
        void stop() override;

        int configure(const IPAConfigInfo &ipaConfig,
                      const std::map<uint32_t, IPAStream> &streamConfig,
                      ControlInfoMap *ipaControls) override;
        void mapBuffers(const std::vector<IPABuffer> &buffers) override;
        void unmapBuffers(const std::vector<unsigned int> &ids) override;

        void queueRequest(const uint32_t frame, const ControlList &controls) override;
        void fillParamsBuffer(const uint32_t frame, const uint32_t bufferId) override;
        void processStatsBuffer(const uint32_t frame, const uint32_t bufferId,
                                const ControlList &sensorControls) override;
};

}
```



src/ipa/brcli/brcli.cpp

The IPA Module

The Algorithms

IDEAS ON BOARD

1. The IPA Interface
2. The Pipeline Handler
3. The IPA Module
4. The Algorithms

- Process the statistics and user controls
- Compute the sensor and ISP parameters

The Algorithms

It’s all maths, really.

The Algorithms

It’s all maths, really.

How hard can it be?

The Algorithms

It’s all maths, really.

How hard can it be?

Maths operations that need to be scheduled.

The Algorithms

It’s all maths, really.

How hard can it be?

Maths operations that need to be scheduled.

In the right order.

The Algorithms

It’s all maths, really.

How hard can it be?

Maths operations that need to be scheduled.

In the right order.
At the right time.

It’s all maths, really.

How hard can it be?

Maths operations that need to be scheduled.

In the right order.
At the right time.
For the right frame.

It’s all maths, really.

How hard can it be?

Maths operations that need to be scheduled.

In the right order.
At the right time.
For the right frame.
In real time.

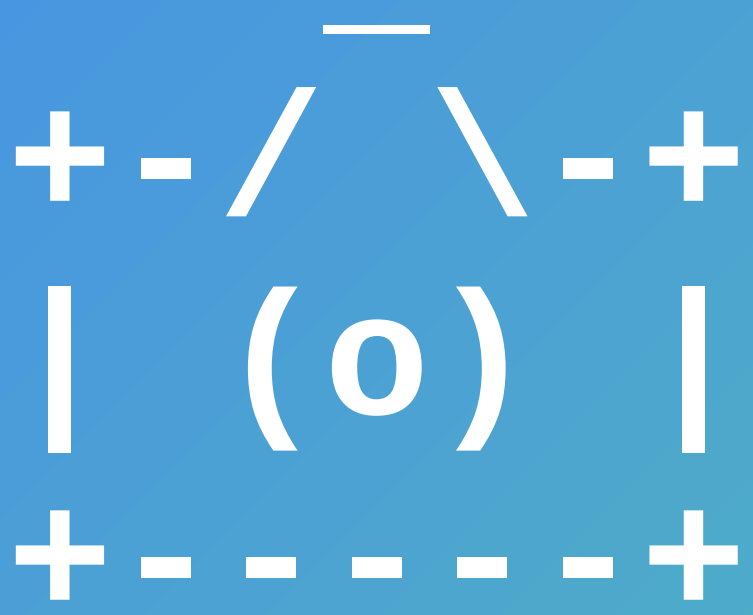
It’s all maths, really.

~~How hard can it be?~~

Maths operations that need to be scheduled.

In the right order.
At the right time.
For the right frame.
In real time.

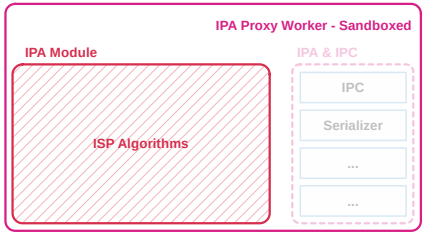
HEEEELP!!!



libipa

```
namespace ipa {

template<typename _Module>
class Algorithm
{
public:
        virtual int init(typename Module::Context &context,
                         const YamlObject &tuningData),
        virtual int configure(typename Module::Context &context,
                              const typename Module::Config &configInfo),
        virtual void queueRequest(typename Module::Context &context,
                                  const uint32_t frame,
                                  typename Module::FrameContext &frameContext,
                                  const ControlList &controls),
        virtual void prepare(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             typename Module::Params *params),
        virtual void process(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             const typename Module::Stats *stats,
                             ControlList &metadata),
};

}
```

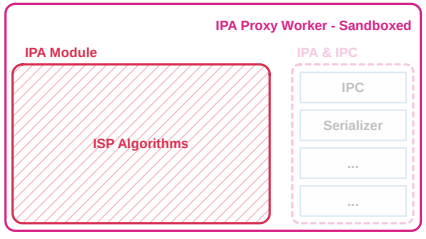
The Algorithm class template defines a common interface for all algorithms.



src/ipa/libipa/algorithm.h

```
namespace ipa {

template<typename _Module>
class Algorithm
{
public:
        virtual int init(typename Module::Context &context,
                         const YamlObject &tuningData),
        virtual int configure(typename Module::Context &context,
                              const typename Module::Config &configInfo),
        virtual void queueRequest(typename Module::Context &context,
                                  const uint32_t frame,
                                  typename Module::FrameContext &frameContext,
                                  const ControlList &controls),
        virtual void prepare(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             typename Module::Params *params),
        virtual void process(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             const typename Module::Stats *stats,
                             ControlList &metadata),
};

}
```

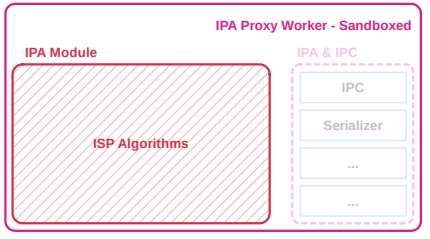
Initialize the context from tuning data. Called once only.



src/ipa/libipa/algorithm.h

```
namespace ipa {

template<typename _Module>
class Algorithm
{
public:
        virtual int init(typename Module::Context &context,
                         const YamlObject &tuningData),
        virtual int configure(typename Module::Context &context,
                              const typename Module::Config &configInfo),
        virtual void queueRequest(typename Module::Context &context,
                                  const uint32_t frame,
                                  typename Module::FrameContext &frameContext,
                                  const ControlList &controls),
        virtual void prepare(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             typename Module::Params *params),
        virtual void process(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             const typename Module::Stats *stats,
                             ControlList &metadata),
};

}
```

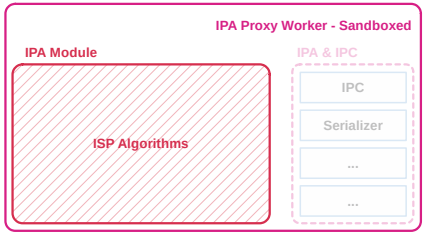
Initialize the algorithm for a capture session, using the camera configuration. Called just before starting capture.



src/ipa/libipa/algorithm.h

```
namespace ipa {

template<typename _Module>
class Algorithm
{
public:
        virtual int init(typename Module::Context &context,
                         const YamlObject &tuningData),
        virtual int configure(typename Module::Context &context,
                              const typename Module::Config &configInfo),
        virtual void queueRequest(typename Module::Context &context,
                                  const uint32_t frame,
                                  typename Module::FrameContext &frameContext,
                                  const ControlList &controls),
        virtual void prepare(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             typename Module::Params *params),
        virtual void process(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             const typename Module::Stats *stats,
                             ControlList &metadata),
};

}
```

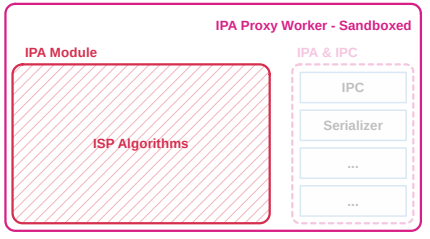
Provide controls values for a frame to the algorithm.



src/ipa/libipa/algorithm.h

```
namespace ipa {

template<typename _Module>
class Algorithm
{
public:
        virtual int init(typename Module::Context &context,
                         const YamlObject &tuningData),
        virtual int configure(typename Module::Context &context,
                              const typename Module::Config &configInfo),
        virtual void queueRequest(typename Module::Context &context,
                                  const uint32_t frame,
                                  typename Module::FrameContext &frameContext,
                                  const ControlList &controls),
        virtual void prepare(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             typename Module::Params *params),
        virtual void process(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             const typename Module::Stats *stats,
                             ControlList &metadata),
};

}
```

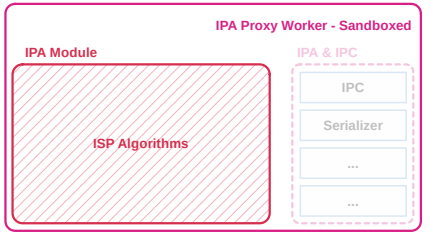
Fill the parameters buffer to prepare ISP processing. Called once per frame.



src/ipa/libipa/algorithm.h

```
namespace ipa {

template<typename _Module>
class Algorithm
{
public:
        virtual int init(typename Module::Context &context,
                         const YamlObject &tuningData),
        virtual int configure(typename Module::Context &context,
                              const typename Module::Config &configInfo),
        virtual void queueRequest(typename Module::Context &context,
                                  const uint32_t frame,
                                  typename Module::FrameContext &frameContext,
                                  const ControlList &controls),
        virtual void prepare(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             typename Module::Params *params),
        virtual void process(typename Module::Context &context,
                             const uint32_t frame,
                             typename Module::FrameContext &frameContext,
                             const typename Module::Stats *stats,
                             ControlList &metadata),
};

}
```

Process ISP statistics. Run the bulk of the algorithm's work. Called once per frame.



src/ipa/libipa/algorithm.h

IPA Algorithms

It’s all maths, really.

~~How hard can it be?~~

Maths operations that need to be scheduled.

In the right order.
At the right time.
For the right frame.
In real time.

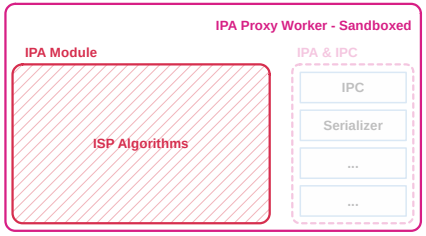
HEEEELP!!!

Thank you!

Algorithms, Take Two

IDEAS ON BOARD

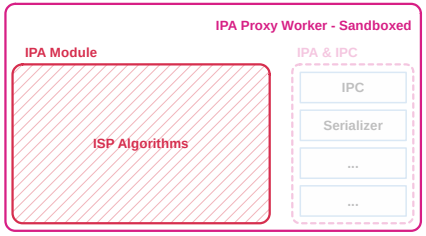
```
namespace ipa::brcli::algorithms {

class BlackLevelCorrection : public Algorithm
{
public:
        BlackLevelCorrection();
        ~BlackLevelCorrection() = default;

        int init(IPAContext &context, const YamlObject &tuningData) override;
        void prepare(IPAContext &context, const uint32_t frame,
                     IPAFrameContext &frameContext,
                     brcli_params_cfg *params) override;

private:
        bool tuningParameters_;
        int16_t blackLevelRed_;
        int16_t blackLevelGreenR_;
        int16_t blackLevelGreenB_;
        int16_t blackLevelBlue_;
};

} /* namespace ipa::brcli::algorithms */
```



src/ipa/brcli/algorithms/blc.h

Algorithms – Black Level Correction

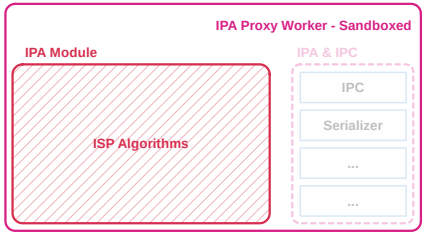
```yaml
# SPDX-License-Identifier: CC0-1.0
%YAML 1.1
---
version: 1
algorithms:
  - Agc:
  - Awb:
  - BlackLevelCorrection:
      R:  256
      Gr: 256
      Gb: 256
      B:  256
```



src/ipa/brcli/data/imx219.yaml

Tuning File

```
int BlackLevelCorrection::init([[maybe_unused]] IPAContext &context,
                               const YamlObject &tuningData)
{
        blackLevelRed_ = tuningData["R"].get<int16_t>(256);
        blackLevelGreenR_ = tuningData["Gr"].get<int16_t>(256);
        blackLevelGreenB_ = tuningData["Gb"].get<int16_t>(256);
        blackLevelBlue_ = tuningData["B"].get<int16_t>(256);

        tuningParameters_ = true;

        LOG(BrcliBlc, Debug)
                << "Black levels: red " << blackLevelRed_
                << ", green (red) " << blackLevelGreenR_
                << ", green (blue) " << blackLevelGreenB_
                << ", blue " << blackLevelBlue_;

        return 0;
}
```

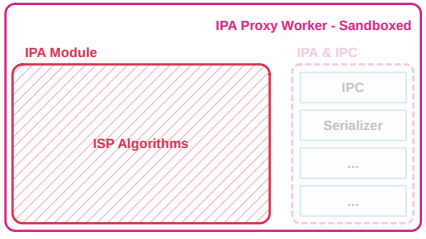


src/ipa/brcli/algorithms/blc.cpp

Algorithms – Black Level Correction

```
void BlackLevelCorrection::prepare([[maybe_unused]] IPAContext &context,
                                   const uint32_t frame,
                                   [[maybe_unused]] IPAFrameContext &frameContext,
                                   brcli_params_cfg *params)
{
        if (frame > 0)
                return;

        if (!tuningParameters_)
                return;

        params->others.bls_config.enable_auto = 0;
        params->others.bls_config.fixed_val.r = blackLevelRed_;
        params->others.bls_config.fixed_val.gr = blackLevelGreenR_;
        params->others.bls_config.fixed_val.gb = blackLevelGreenB_;
        params->others.bls_config.fixed_val.b = blackLevelBlue_;

        params->module_en_update |= BRCLI_CIF_ISP_MODULE_BLS;
        params->module_ens |= BRCLI_CIF_ISP_MODULE_BLS;
        params->module_cfg_update |= BRCLI_CIF_ISP_MODULE_BLS;
}
```

This is where the algorithm interfaces with the ISP kernel driver API.



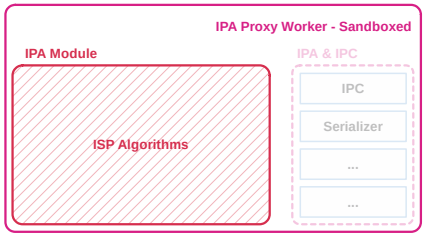
src/ipa/brcli/algorithms/blc.cpp

Algorithms – Black Level Correction

```
REGISTER_IPA_ALGORITHM(BlackLevelCorrection, "BlackLevelCorrection")
```

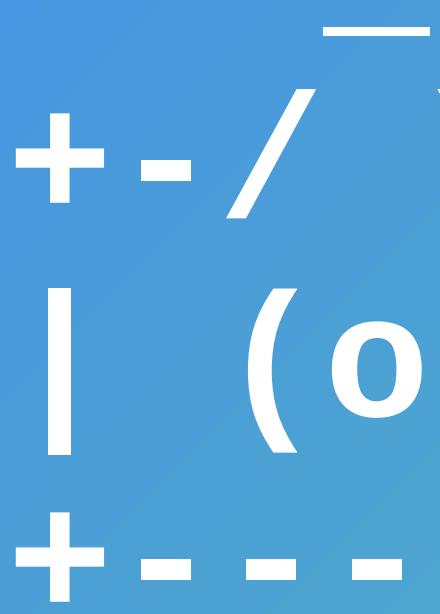
The class name.

The algorithm name in the tuning file.



src/ipa/brcli/algorithms/blc.cpp

Algorithms – Black Level Correction

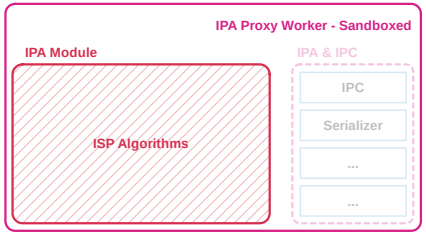


Back To The IPA Module

IDEAS ON BOARD

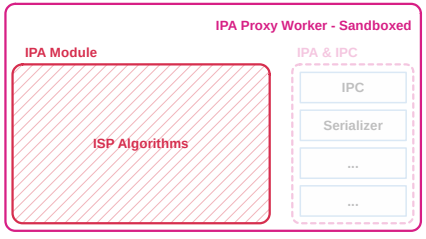
```
namespace ipa {

template<typename _Context, typename _FrameContext, typename _Config,
         typename _Params, typename _Stats>
class Module : public Loggable
{
public:
        const std::list<std::unique_ptr<Algorithm<Module>>> &algorithms() const;

        int createAlgorithms(Context &context, const YamlObject &algorithms);

        static void registerAlgorithm(AlgorithmFactoryBase<Module> *factory);
};

}
```

The libipa Module class template handles algorithms management for IPA modules.



src/ipa/libipa/module.h

```
namespace ipa::brcli {

using Module = ipa::Module<IPAContext, IPAFrameContext, IPACameraSensorInfo,
                           brcli_params_cfg, brcli_stat_buffer>;

class IPABrcli : public IPABrcliInterface, public Module
{
public:
        IPABrcli();

        int init(const IPASettings &settings, unsigned int hwRevision,
                 const IPACameraSensorInfo &sensorInfo,
                 const ControlInfoMap &sensorControls,
                 ControlInfoMap *ipaControls) override;
        int start() override;
        void stop() override;

        int configure(const IPAConfigInfo &ipaConfig,
                      const std::map<uint32_t, IPAStream> &streamConfig,
                      ControlInfoMap *ipaControls) override;
...
};

}
```

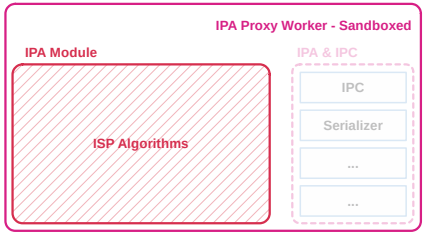


src/ipa/brcli/brcli.cpp

The IPA Module – With Help

```
int IPABrcli::init(const IPASettings &settings, unsigned int hwRevision,
                   const IPACameraSensorInfo &sensorInfo,
                   const ControlInfoMap &sensorControls,
                   ControlInfoMap *ipaControls)
{
...
        File file(settings.configurationFile);
        file.open(File::OpenModeFlag::ReadOnly));

        std::unique_ptr<libcamera::YamlObject> data = YamlParser::parse(file);
        if (!data)
                return -EINVAL;

        if (!data->contains("algorithms")) {
                LOG(IPABrcli, Error)
                        << "Tuning file doesn't contain any algorithm";
                return -EINVAL;
        }

        createAlgorithms(context_, (*data)["algorithms"]);
...
}
```

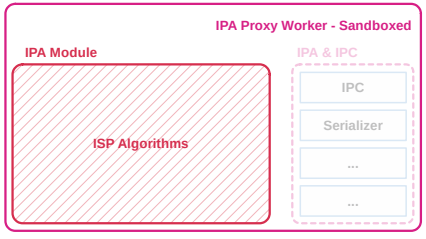
Algorithms are instantiated from the tuning data file.



src/ipa/brcli/brcli.cpp

The IPA Module – With Help

```
int IPABrcli::configure(const IPAConfigInfo &ipaConfig,
                        const std::map<uint32_t, IPAStream> &streamConfig,
                        ControlInfoMap *ipaControls)
{
...
        for (auto const &a : algorithms()) {
                Algorithm *algo = static_cast<Algorithm *>(a.get());

                /* Disable algorithms that don't support raw formats. */
                algo->disabled_ = context_.configuration.raw && !algo->supportsRaw_;
                if (algo->disabled_)
                        continue;

                int ret = algo->configure(context_, info);
                if (ret)
                        return ret;
        }
...
}
```

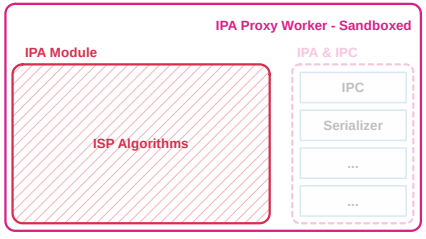


src/ipa/brcli/brcli.cpp

The IPA Module – With Help

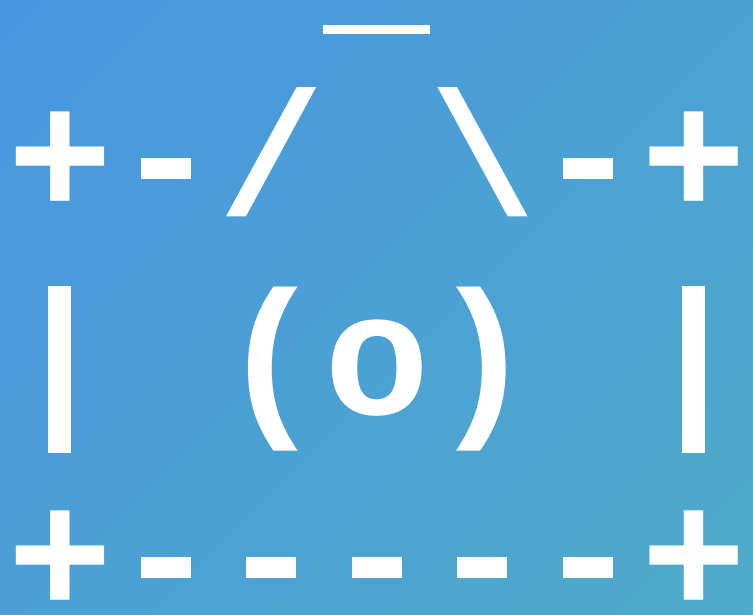
```
void IPABrcli::queueRequest(const uint32_t frame, const ControlList &controls)
{
        IPAFrameContext &frameContext = context_.frameContexts.alloc(frame);

        for (auto const &a : algorithms()) {
                Algorithm *algo = static_cast<Algorithm *>(a.get());
                if (algo->disabled_)
                        continue;
                algo->queueRequest(context_, frame, frameContext, controls);
        }
}
```

Similarly, fillParamsBuffer() calls algo->prepare(), and processStatsBuffer() calls algo->process().



src/ipa/brcli/brcli.cpp

The IPA Module – With Help



libcamera

IDEAS ON BOARD

libcamera-devel@lists.libcamera.org
ircs://irc.oftc.net/#libcamera

laurent.pinchart@ideasonboard.com

Contact

By the way, we are hiring
jobs@ideasonboard.com

Merci