

The slide features a blue-to-teal gradient background with a subtle diagonal line pattern. On the left, there is a white ASCII art logo that resembles a face, composed of symbols like '+', '-', '/', '\', '|', and 'o'. Below this logo is an orange badge with the text 'IDEAS ON BOARD'. To the right of the logo, the title 'Giving Linux a Camera Stack: libcamera's 3 Years Journey and Exciting Future' is written in white. Below the title, the event information 'Embedded Linux Conference 2021' and 'Seattle, 2021-09-28' is displayed. At the bottom right, the speaker's name 'Laurent Pinchart' and email 'laurent.pinchart@ideasonboard.com' are listed.

Giving Linux a Camera Stack: libcamera's 3 Years Journey and Exciting Future

Embedded Linux Conference 2021
Seattle, 2021-09-28

Laurent Pinchart
laurent.pinchart@ideasonboard.com



Hello everybody, and welcome to this presentation about libcamera. For those of you who are lucky enough to join us live from Seattle today, thank you for waking up early. This is the first slot of the day, so I know how difficult it can be.

My name is Laurent Pinchart. I'm the chief architect and project manager of libcamera. Today I'm going to take you on libcamera's fabulous journey.

So let's dive in the subject.

Once Upon A Time



Before we actually start talking about libcamera, I think it's important to understand the context and where camera support in Linux camera from.



Panasonic MC20 + Pinnacle Miro DC10



In the beginning...

What you're seeing on the screen is my very first webcam. It's made of a VHS camcorder connected through a composite video cable to a PCI capture card. It was bulky and required an actual tape inside the camcorder to operate, but it did the job. After months of reverse engineering it even worked on Linux. That was my idea of well spent free time in those days.



Logitech Quickcam Express



... were simple devices

After a brief period of crazy ideas such as connecting cameras through the printer parallel port (which Linux actually supported), at the end of the 20th century the world moved to USB-connected cameras. For all purpose but nostalgia it was a good idea.

*A monolithic API for
TV grabbers and
webcams alike.*



*High-level controls,
the TV signal
provides a good
image already.*

*Enables development of
universal applications.*



V4L2

What did all those devices have in common ? A large portion of them had Linux drivers – mostly developed through reverse-engineering –, and all those drivers implemented the Video4Linux API.

In the rest of this presentation I will use the words Video4Linux, V4L and V4L2 interchangeably, the old V4L1 API is only of interest for historians.

Video4Linux is a broad API with lots of features to accommodate different kinds of video capture devices, from TV grabbers to webcams. It is fairly monolithic, in the sense that it tries to apply the same model to all those devices. As a result, an application that supports V4L2 will likely work with TV grabbers and webcams alike, at least for their basic features.

V4L is also a fairly high-level API because it maps directly to the features of the devices it supports. My camcorder handled the gory details of auto-exposure and auto-white balance, and USB webcams followed the same model. V4L didn't have to care about what happened under the hood inside the cameras.



Logitech Quickcam Express For Notebooks



Early UVC Camera

What happened next ? Fast forward 5 years, the industry standardized on a common USB protocol for webcams (2003), called the USB Video Class (UVC).

From: Laurent Pinchart <laurent.pinchart@skynet.be>
To: linux-usb@vger.kernel.org, video4linux-list@redhat.com
Subject: [PATCH] USB Video Class driver
Date: Wed, 23 Apr 2008 01:37:11 +0200

Hi everybody,

after more than two years of development the Linux UVC driver is mostly ready to jump the fence and get included in the mainline kernel.

This driver aims to support video input devices compliant with the USB Video Class specification. This means lots of currently manufactured webcams, and probably most of the future ones.

I plan to submit the driver through the V4L subsystem, but I'd like it to get a proper review on both the linux-usb and video4linux mailing lists first.

Given the size of the patch I'm open to any suggestion that would make the review process easier.

Laurent Pinchart

(my humble 25 August 1991 moment)



USB Video Class

It took two years to develop a Linux driver for it. This was my humble first real steps in the Linux kernel world, and got merged in 2008.

As virtually all new webcams were expected to implement the USB Video Class, I thought camera support in Linux was solved for good. The future proved me sooo wrong.



Palm Treo 650 (PXA270, 0.3MP Camera)



Embedded Cameras

What did I miss ? First of all, not all cameras were connected over USB. Linux was gaining fast adoption in the embedded world, and there the situation was different.

From: Guennadi Liakhovetski <g.liakhovetski@pengutronix.de>
To: video4linux-list@redhat.com
Subject: [PATCH 1/6] soc_camera V4L2 driver for directly-connected
SoC-based cameras
Date: Tue, 5 Feb 2008 18:46:13 +0100 (CET)

This driver provides an interface between platform-specific camera busses and camera devices. It should be used if the camera is connected not over a "proper" bus like PCI or USB, but over a special bus, like, for example, the Quick Capture interface on PXA270 SoCs. Later it should also be used for i.MX31 SoCs from Freescale. It can handle multiple cameras and / or multiple busses, which can be used, e.g., in stereo-vision applications.

Signed-off-by: Guennadi Liakhovetski <g.liakhovetski@pengutronix.de>



V4L2 Goes Embedded

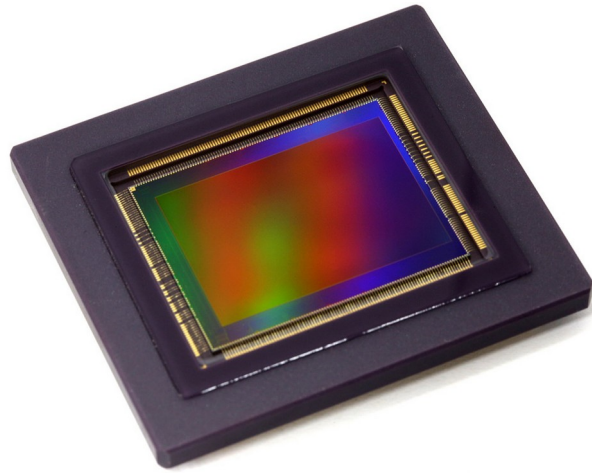
In embedded devices, cameras are made of an image sensor that outputs data on a dedicated hardware interface, such as MIPI CSI-2 in today's devices. The SoC, the processor running Linux, integrates a receiver compatible with that interface, which then transfers images to memory using DMA.

Unlike in webcams, both the camera sensor and the receiver are directly controlled by Linux.

What is a camera?

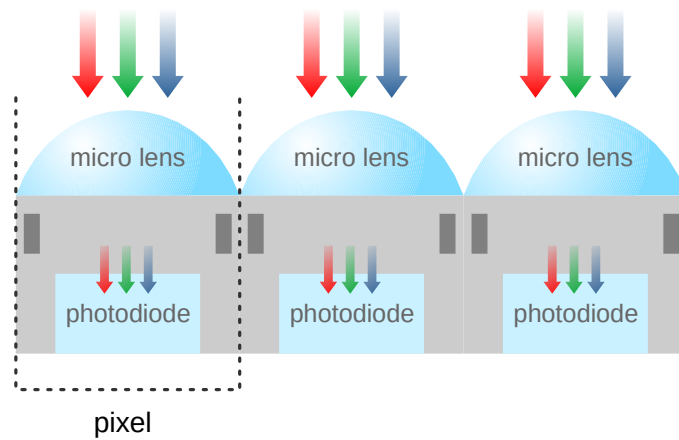


What implications does this have ? To answer that question, we need to understand what a camera actually is.



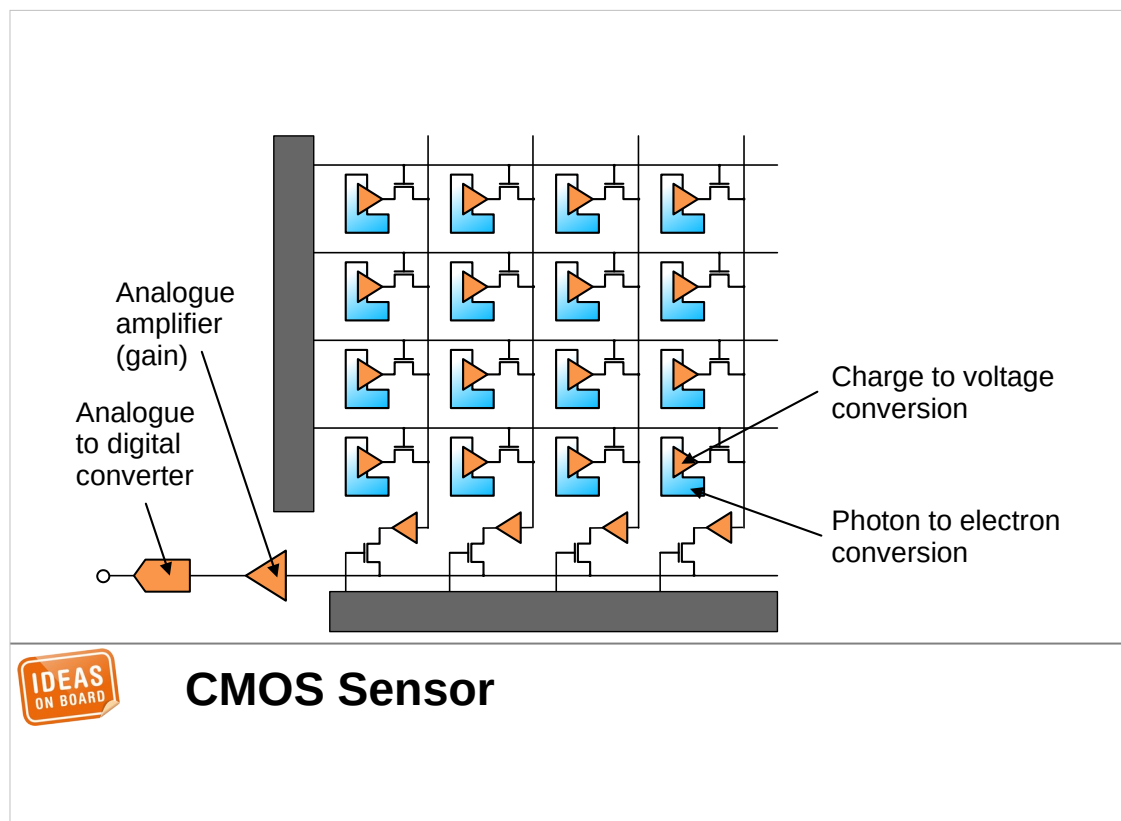
CMOS Sensor

At the core of a camera, as you likely all expect, is an imaging sensor. This is a device that converts light to digital values.



CMOS Sensor

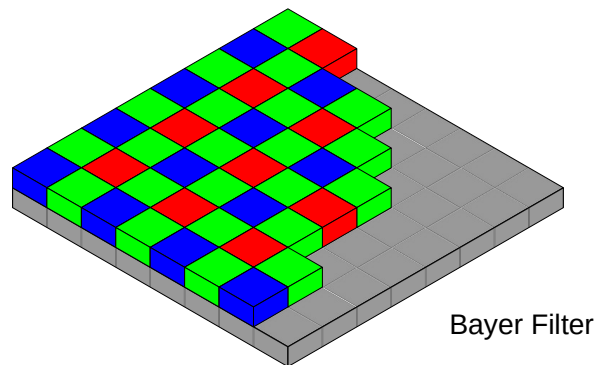
It is made of tiny light-sensitive photodiodes with the electronics required to convert the charge into a voltage. Each of them has a micro-lens to gather as much light as possible into the diode. Those are what we call pixels.



The pixels are assembled into an array, with rows and columns, and additional electronics to route the voltage of each pixel one after the other to an analogue amplifier and an analogue to digital converter.

The sensor outputs a digital value for each pixel proportional to the amount of light that has reached the photodiode. This is our first trouble: the diodes are sensitive to an amount of light, so they produce a greyscale image.

How do we get colours ?



Colour Filter Array

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

Colours are related to physical properties of the light and its spectral contents, but it's important to realize here that the concept of colour is deeply tied to the human eye and its perception of light. Without going into details, let's just remember that colour processing in cameras is mostly about outputting images that appear as realistic as possible to the human eye.

What most camera sensors do is simply put tiny colour filters in front of each pixels, with colours corresponding to the sensitivity of the human eye to red, green and blue. The most common arrangement of such filters groups them in cells of 2 by 2 pixels with one red, one blue and two green filters, because the eye is more sensitive to green than red or blue. This colour filter array pattern is called Bayer.



Original



120x80 Bayer image



CFA Interpolation

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

Let's look at the impact this has on the image. Image 1 shows a sample scene that we then capture with a 120x80 pixels Bayer sensor.

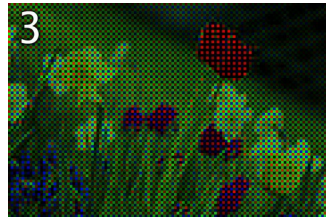
Image 2 shows the value of all captured pixels. As you can see from the red flower, we have a checker pattern of light and dark pixels. The light pixels correspond to the ones with a red filter, which lets the red light through, while the dark pixels correspond to the green and blue filters, which block most of the red light.



Original



120x80 Bayer image



Colour-coded



CFA Interpolation

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

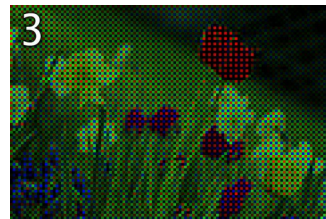
Image 3 shows the same image, but with each pixel colourised with the colour of the corresponding filter. We can see that colour information is present, but each pixel misses two out of the three colour channels. This leads to the typical checker pattern of colourised Bayer images.



Original



120x80 Bayer image



Colour-coded



Colour interpolation



CFA Interpolation

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

Image 4 is obtained by interpolating the missing colour components using the values of neighbouring pixels. For instance, a red pixel is surrounded by 4 green pixels, its green value can be estimated by taking the average of those 4 green neighbours. In practice, to obtain a good image quality, more complex interpolation is required, taking into account more neighbours with more complex mathematical operations.

It would be easy if it stopped there.

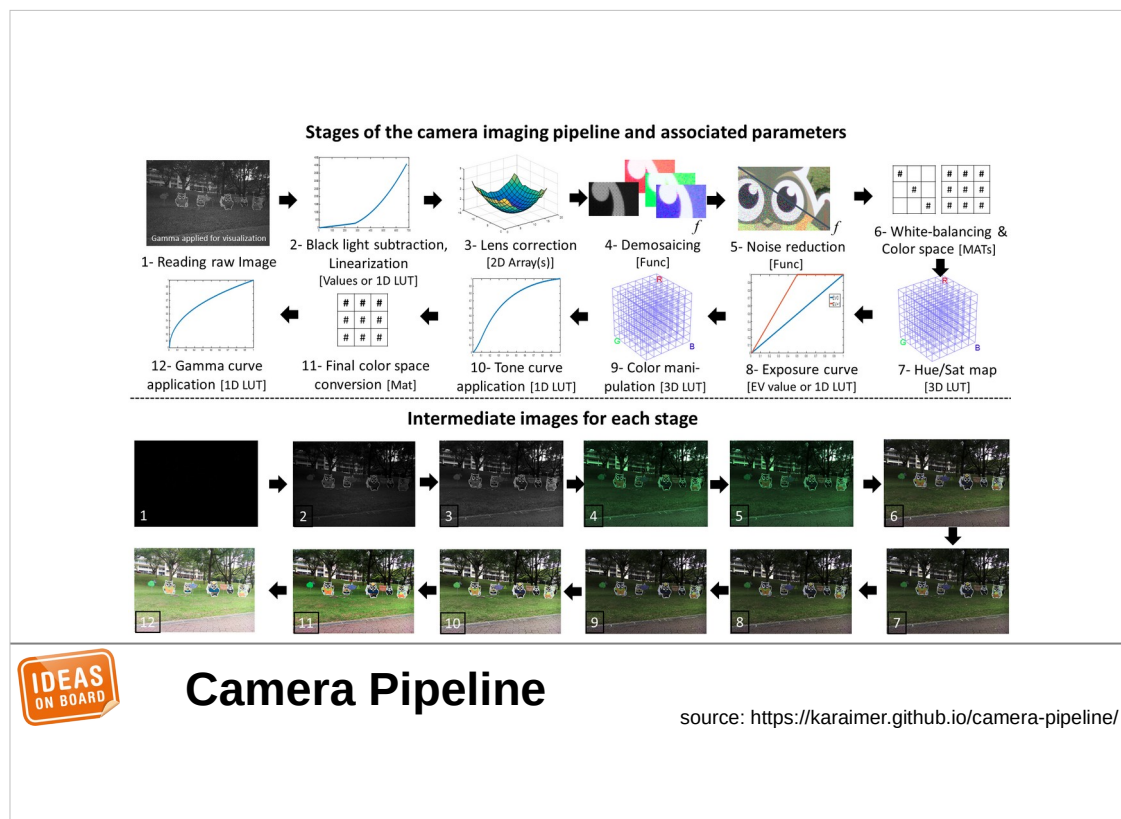


Lens Shading

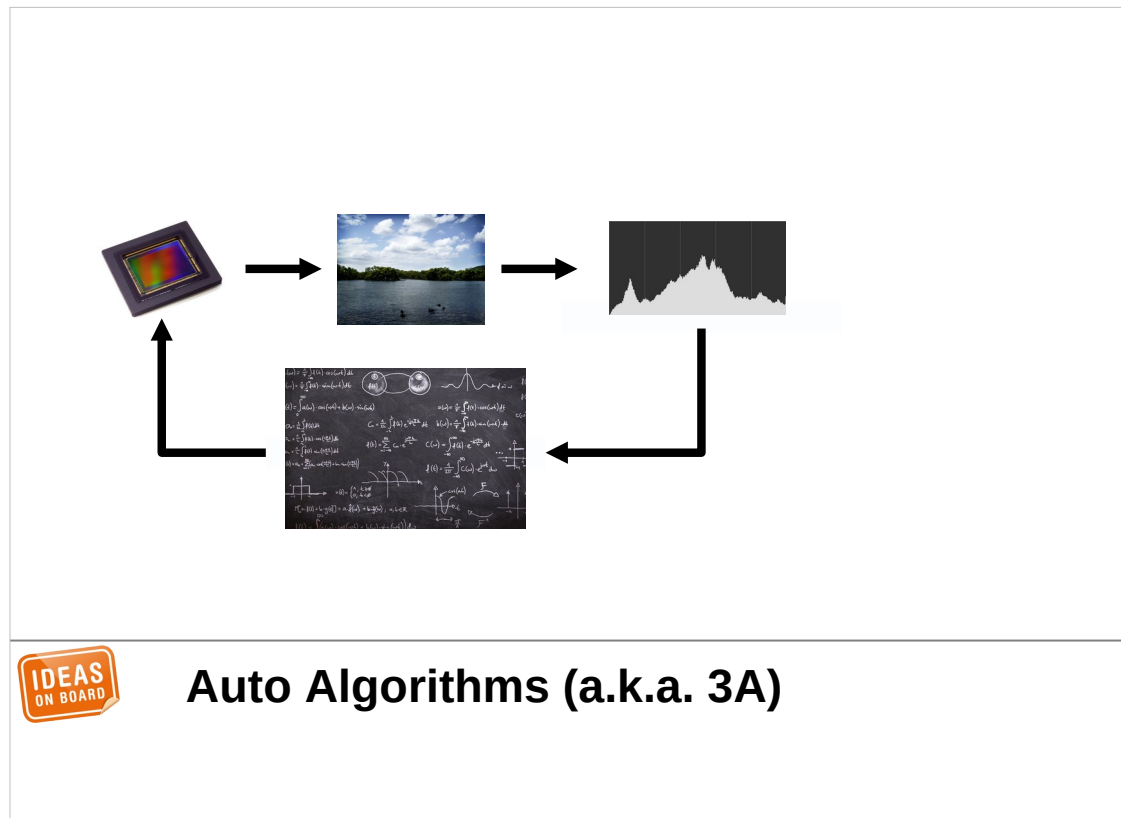
There is more processing that needs to be applied to the image to achieve an acceptable quality. This is caused in part by imperfections in the optics and camera sensor.

For instance, as shown here, the lens lets less light through on the periphery than in the centre, causing an undesired vignetting effect.

There are plenty of other issues. The sensor may have defective pixels that need to be hidden. The total absence of light isn't rendered black due to leakage currents in the photodiode. Noise affects the image at all stages from the photodiode to the analogue to digital conversion. The list goes on.



Images thus need to go through a complex camera pipeline, way too expensive to implement in software in real-time. Cameras need hardware assistance, and this is provided by specialized devices called Image Signal Processors (ISP).



And if you thought that was complex enough, it's not all.

The luminance of the scene, in front of the camera, typically varies constantly. This requires adjusting the integration time and gain of the sensor accordingly, to produce an image this is neither underexposed nor overexposed. The same is true for the white balance, which requires adjusting colour gains based on the light of the scene, or for the focus as people can move in front of the camera.

Parameters that control the lens, the sensor and the ISP need to be computed in real time, based on an analysis of the captured images. This is again computationally-intensive, but fortunately the ISP comes to the rescue by computing statistics such as histograms. Based on those statistics, algorithms compute the processing parameters for the next frame, and apply them to the device. The same process repeats for every frame.

Those algorithms are often referred to as 3A for auto-exposure, auto-white balance and auto-focus, but can include more processing. All of this is key to getting the final quality of the image.



IQ Tuning

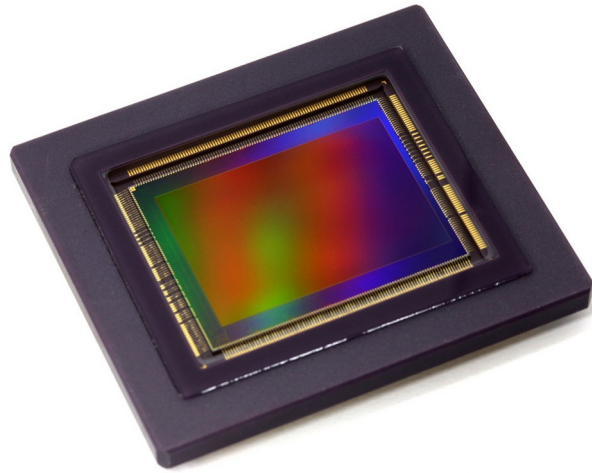
source: <https://www.flickr.com/photos/davedugdale/15043975135>

Last but not least, the algorithms need to be calibrated and tuned for every combination of a camera sensor and optics. This adds even more complexity to the development process.

Back To V4L2



Now, do we really have to implement all this for embedded cameras ?



Smart Sensor (a.k.a. YUV Sensor)

Fortunately for us, there are sensors that embed an ISP and a microcontroller to run the algorithms. They are called smart sensors, or YUV sensors because they output processed images in YUV format as opposed to raw unprocessed Bayer images.

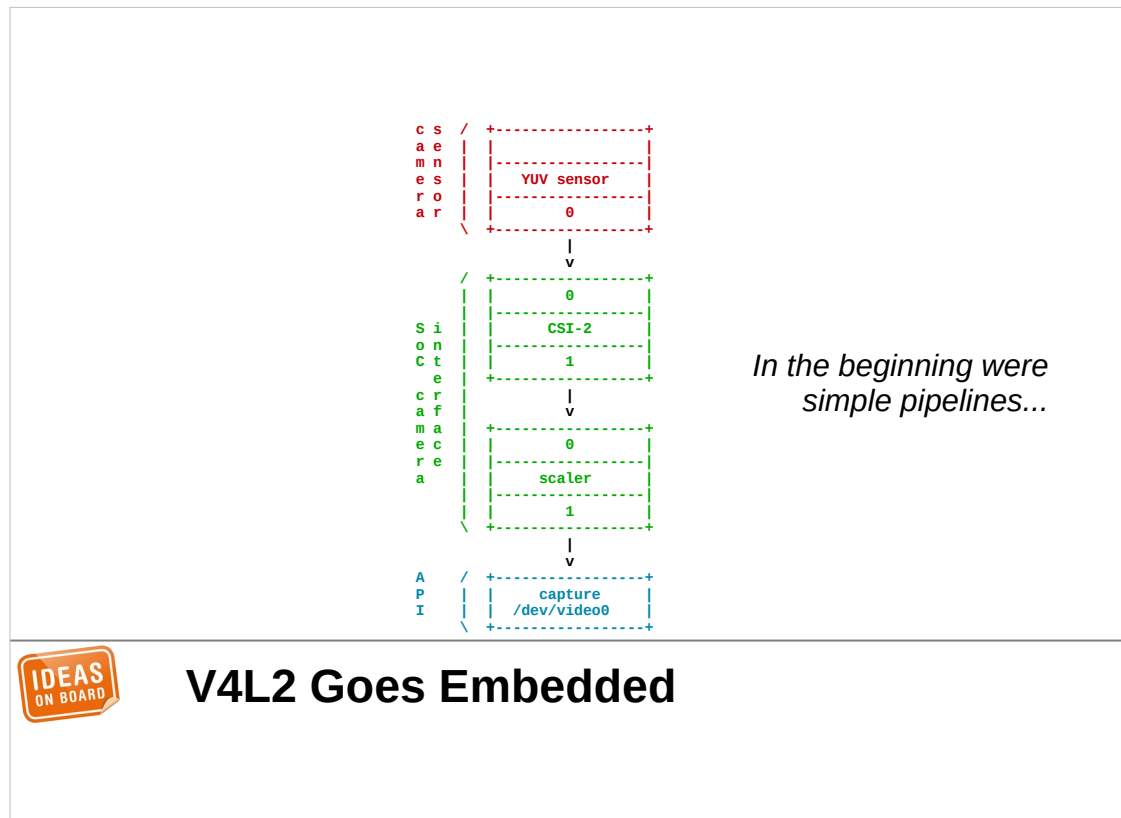


Logitech Quickcam Express For Notebooks



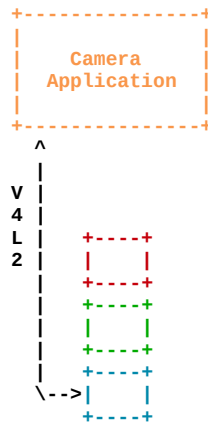
So Small, Yet So Smart

On a side note, USB webcams often use smart sensors, and when they don't, they integrate a separate ISP in the camera. The integrated webcams in laptops are also USB devices, that's why Linux didn't have to care so far.



The day is saved, we can use our smart sensor connected to the SoC. There is a bit more to deal with than with a USB webcam, as the camera receiver in the SoC may have additional features, such as the ability to scale. Still, the complexity of the camera pipeline is very limited.

*... and they were
simple to control,
with a single API.*



V4L2 Goes Embedded

And it can all be exposed through V4L2, which supports scaling already. Existing V4L2 applications can keep using the same API, unaware that the SoC has a scaler separate from the camera sensor. Everything works like if the embedded camera was a webcam.



Then the world became complex

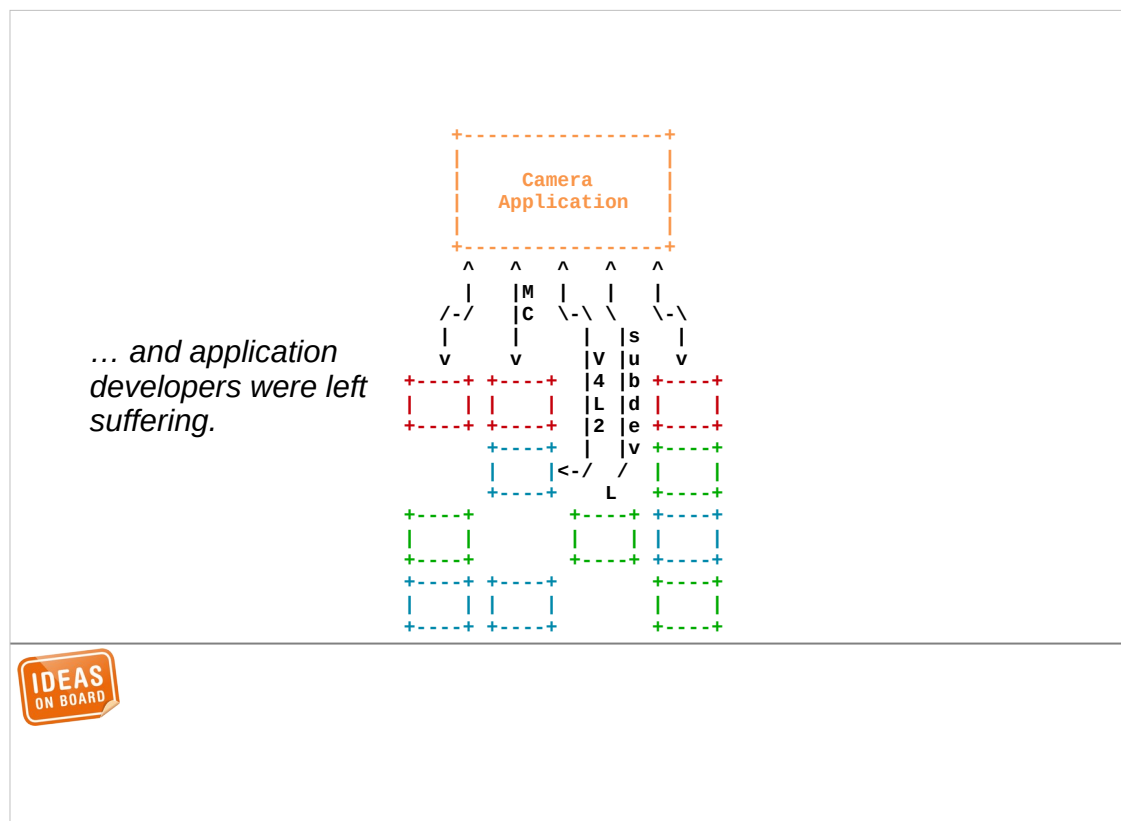
Then trouble happened. This is the Nokia N900, one of the first Linux phones from a large manufacturer. It is special because it marks the beginning of the libcamera journey.

+ - / \ - +
| (o) |
+ - - - - +

The libcamera Journey



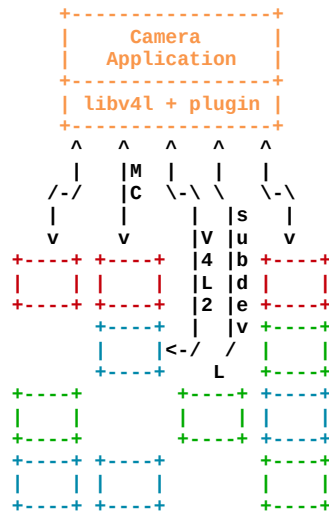
You will ask me, how can a phone released in 2009 mark the beginning of libcamera, which was announced nearly 10 years later, in 2018 ?



This is all nice for the kernel, but not for applications. They had to program sensor and ISP parameters, capture raw frames, pass the raw frames to the ISP, capture processed frames and statistics, and implement the image processing algorithms. All this is also device-dependent, as different ISPs behave differently and algorithms can't be generalized. The idea of portable V4L2 applications that would work with different cameras got completely shattered at that point.

Nokia had the resources to develop a custom camera stack in userspace (which was partly proprietary) and custom applications. This was beyond the reach of most developers, whether hobbyists or working for small to medium-size companies. The complexity was just not manageable.

*Solutions were
proposed...*



We had envisioned solutions to this problem, with designs based on platform-specific plugins for libv4l, the V4L2 wrapper library.



*... but never
implemented.*



Unfortunately, on February 11th, 2011, Nokia decided to cancel its line of Linux-based mobile phones and switch to Windows Phone. Development of userspace solutions stopped.



From 2011, Linux was without an embedded camera stack. Development continued on the kernel side, but nobody could or would commit enough resources to fix the situation in userspace.



Acer Chromebook Tab 10

Meanwhile, new devices got developed with raw camera sensors and an ISP in the SoC. This architecture was spreading from phones to ARM-based tablets...

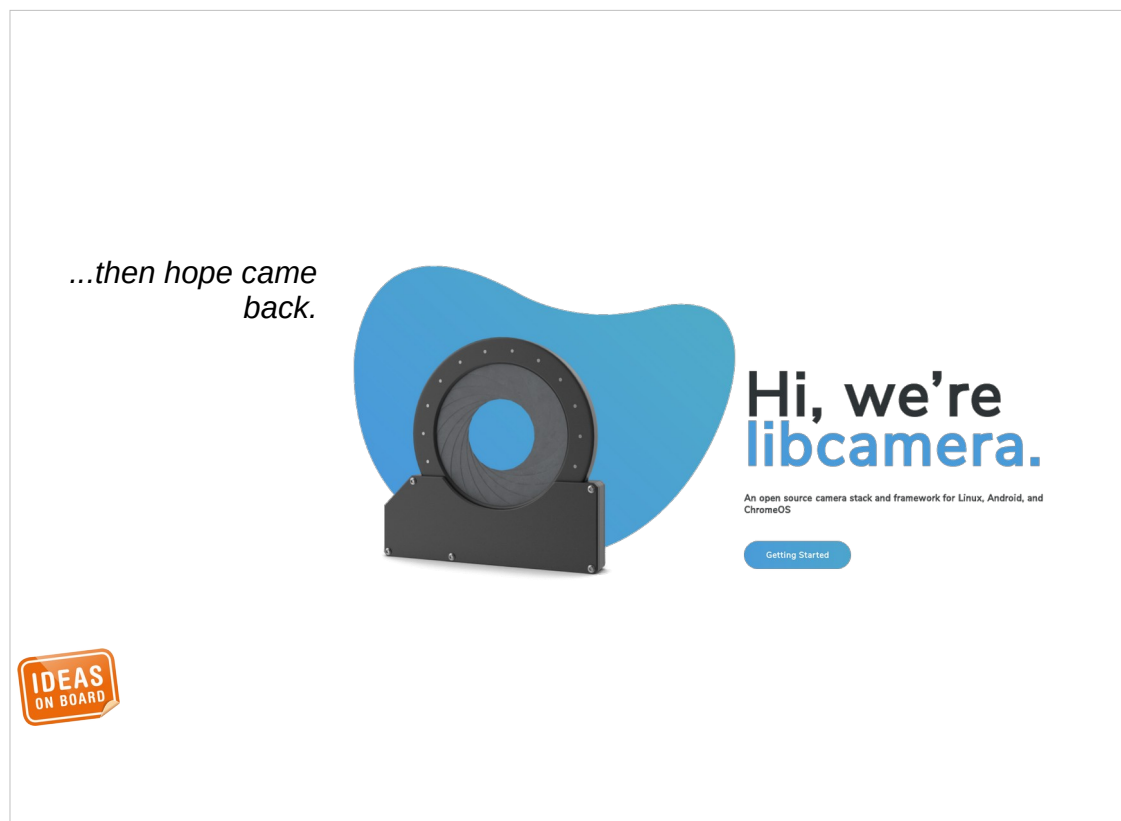


... and from tablets to laptops, even on Intel-based devices. This particular laptop uses a raw sensor with an Intel Kaby Lake SoC.

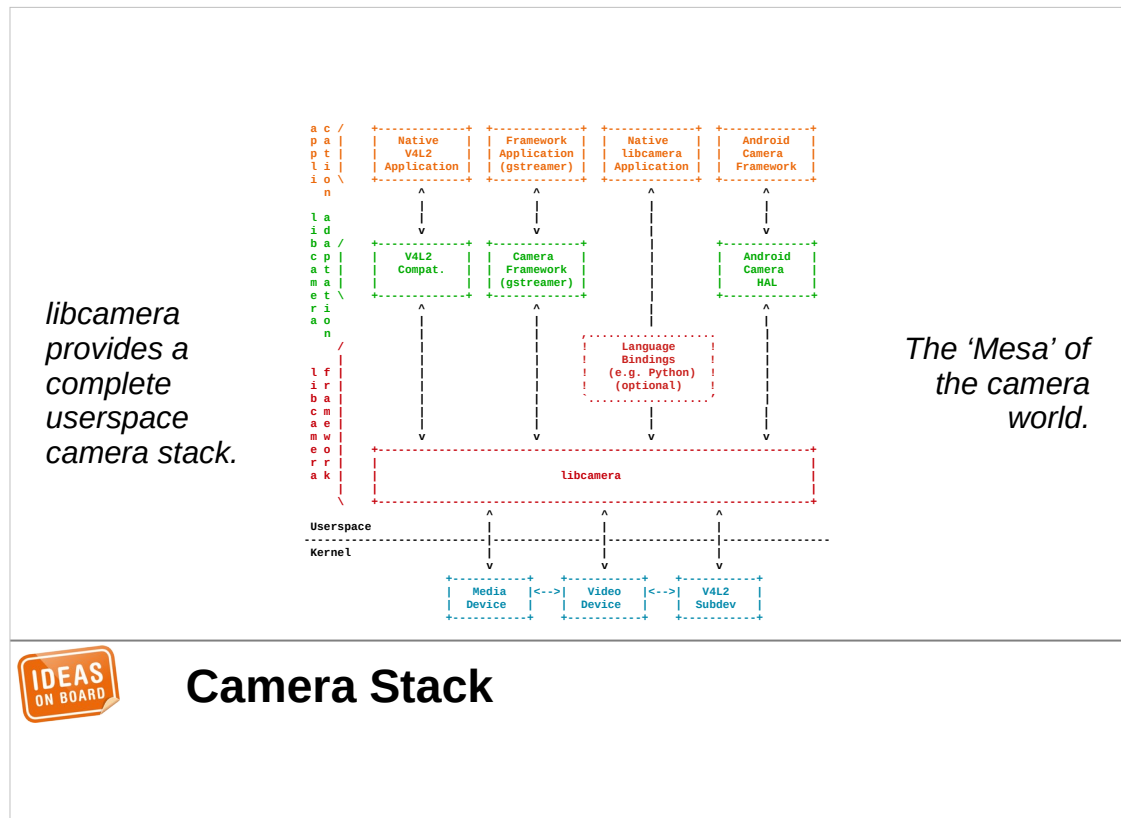
Why did vendors decide to use raw sensors, and not smart sensors or USB webcams integrated in the device case ? Was it pure masochism ? It turns out there are multiple reasons. They are related to cost, size (USB webcam modules can be quite thick), but most importantly to image quality and features. With raw sensors, no silicon space is used to implement an ISP, and we can achieve larger pixels sizes and resolutions. With a separate controllable ISP, vendors can implement more advanced image algorithms. A couple of very simple examples are focus assistance based on face detection, or advanced HDR processing. Many more complex use cases exist.

As those features can be fairly advanced, they are often considered by vendors as a key differentiating factor that needs to be covered by the uttermost secrecy. This makes embedded cameras and free software unlikely friends.

Regardless of the reason, the trend was clear, and was here to stay. The problem had to be addressed urgently.



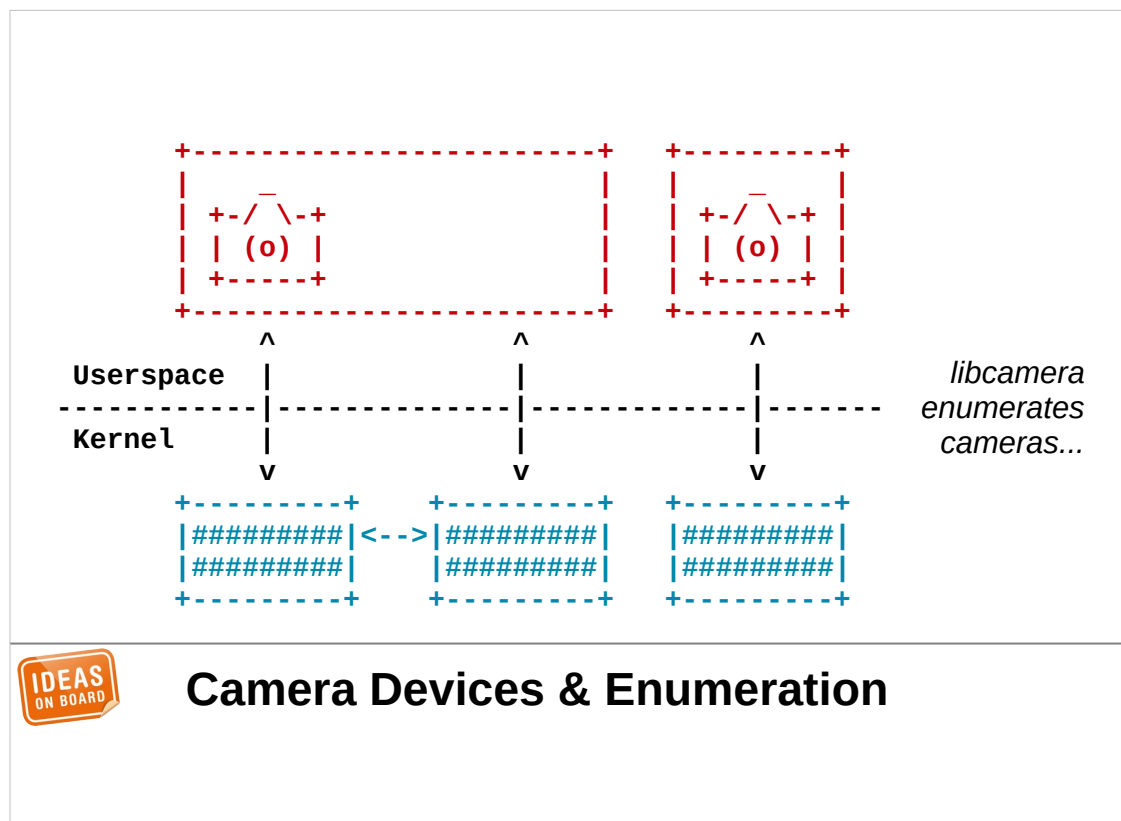
At the end of 2018, after contacts with the industry over the summer, we announced the libcamera project.



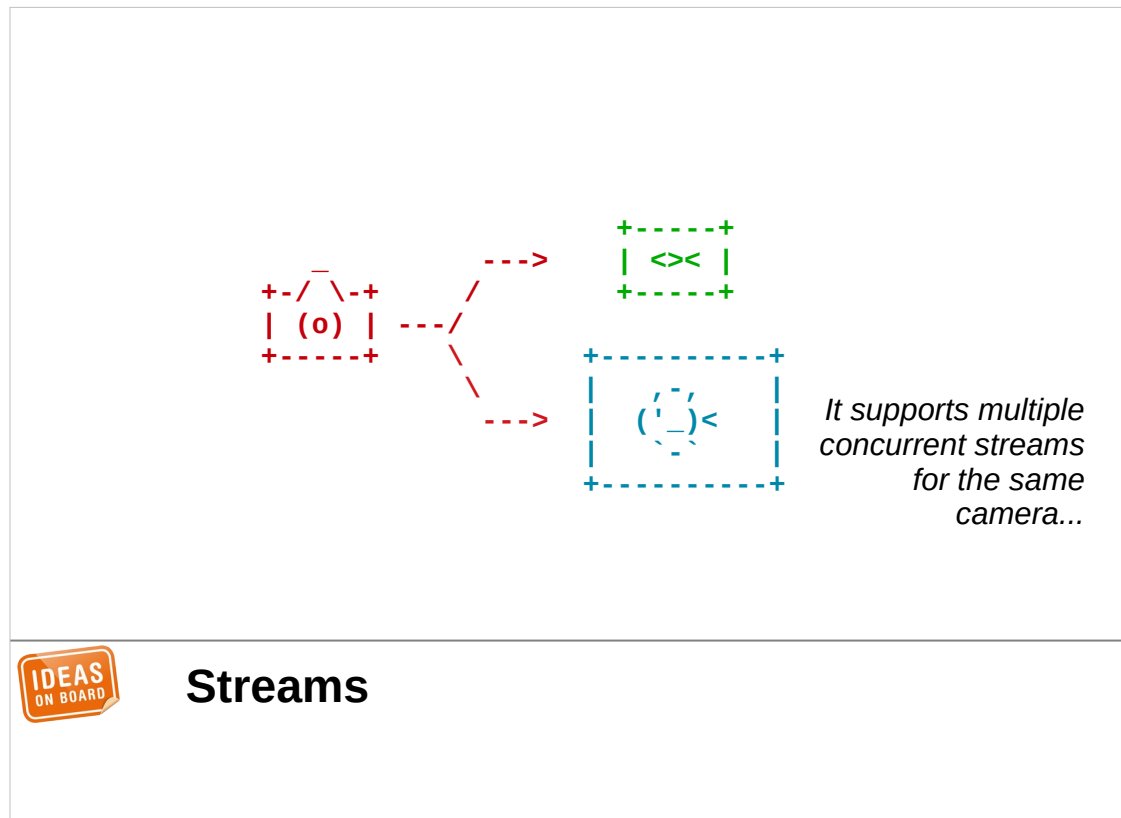
The goals were ambitious. libcamera was to provide a complete userspace camera stack, with a new native API, and a feature set that would at least match the capabilities of the Android camera API. This was way beyond what V4L2 supports natively.

And of course it had to be free software.

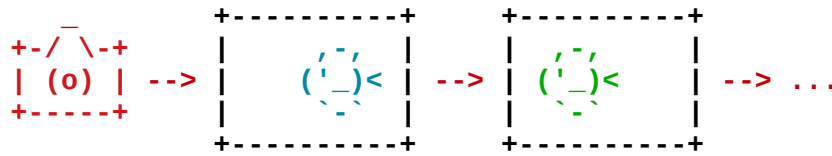
While our initial team was actually coming from kernel development, we wanted to focus our attention on the userspace side as much as possible. libcamera thus uses existing V4L2 and MC kernel drivers, it doesn't come with a new kernel API.



libcamera had to handle camera enumeration and support multiple cameras concurrently, ...



... and also support multiple concurrent streams per camera, to capture the same frame in different resolutions and formats. This way, an application can for instance obtain a stream with the native screen resolution to display locally, and a second stream with a different resolution for video recording or streaming.



... and per-frame controls.



Per-Frame Controls

libcamera would support per-frame controls. It would guarantee to applications that control parameters get applied precisely to the requested frame.

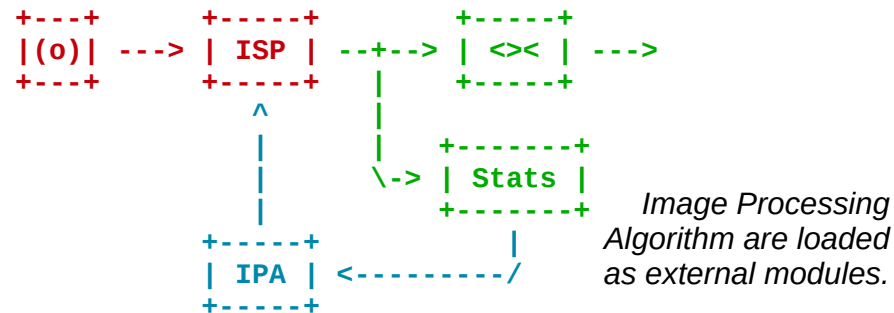
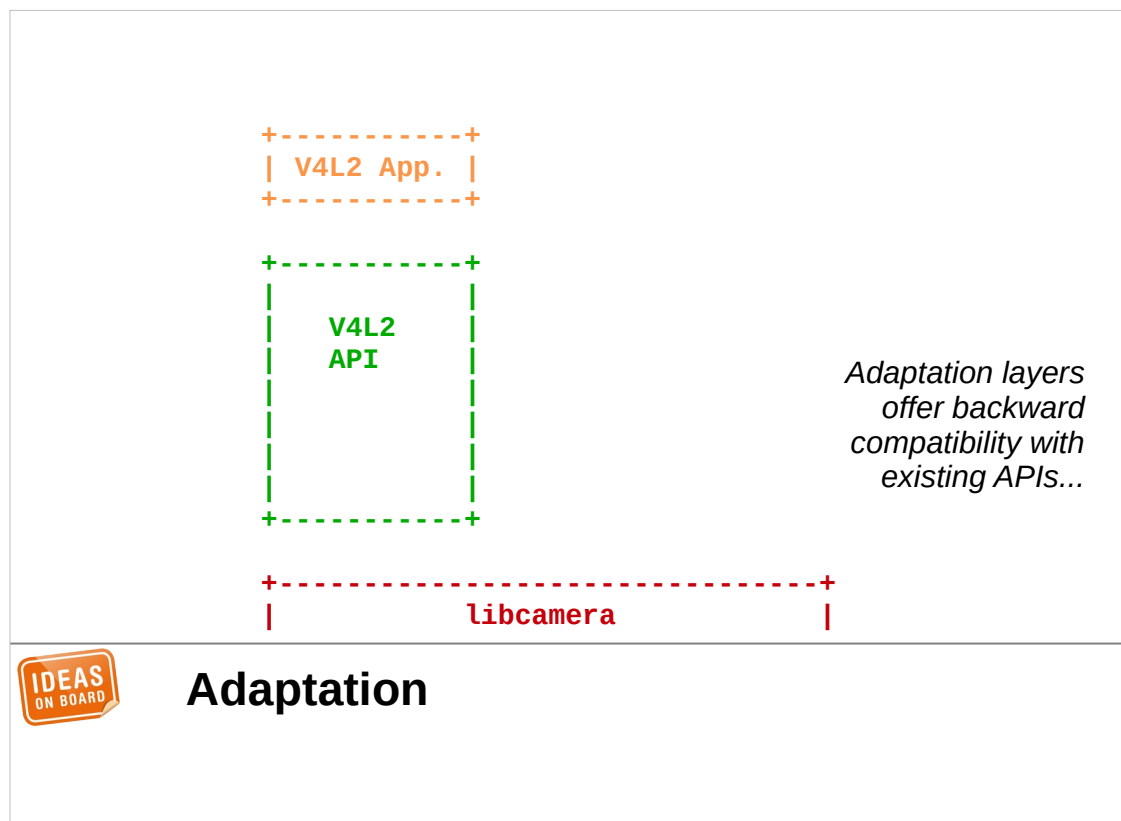


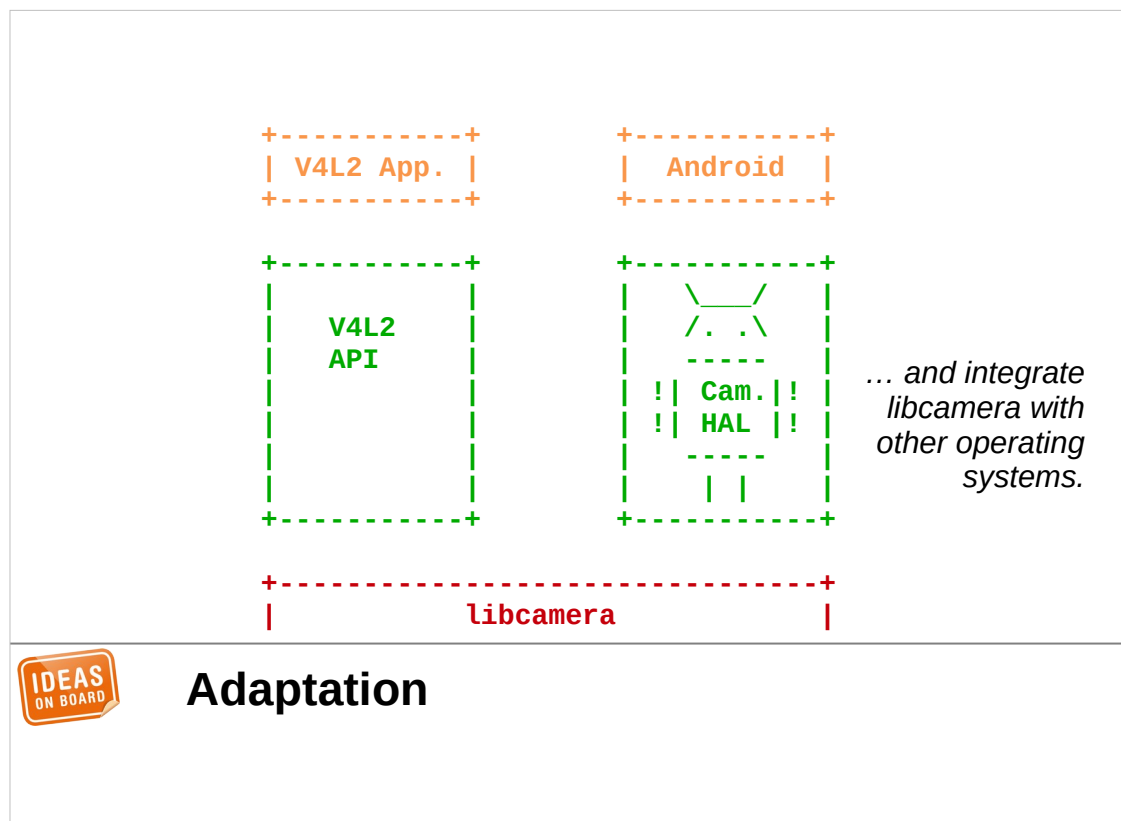
Image Processing Algorithms (3A)

Of course, as libcamera had to control ISPs, it would need to implement image processing algorithms. We have seen how those algorithms are some of the most protected vendor IP, so we decided to isolate them in plugins named Image Processing Algorithm modules, or IPA modules.

This architecture allowed vendors to provide closed-source implementations of algorithms that can coexist with open-source implementations, even for the same platform. We will see later how this was done without compromising on the ability to use cameras with free software only.



Even though libcamera offers a native API, right from the beginning we considered the need to be backward-compatible with existing APIs to facilitate libcamera's adoption. In particular, we wanted libcamera to be usable with most V4L2 applications without having to even recompile them. We will also see later how that was achieved.



We also considered as a main goal support for Android and Chrome OS, which are both based on the same Android camera API.

For those who are not familiar with camera implementation on Android, Android defines an API for camera provider modules named the camera Hardware Abstraction Layer, or HAL. They require device vendors to provide a camera HAL implementation, and Android builds the camera service on top. With a single implementation of the Android camera HAL based on libcamera, we can support both operating systems, Android and Chrome OS.

+ - / \ - +
| (o) |
+ - - - - +

The libcamera Journey



So at end of 2018, we had a goal, an architecture, and plenty of motivation. The adventure could begin.



Intel IPU3 (Kaby Lake)
on HP Chromebook x2



USB Video Class (UVC)



Our Initial Targets

We started development by targetting two very different devices initially. Our main goal was the ISP found in Intel Kaby Lake SoCs, named IPU3. We picked a Chromebook device as a development platform, as it had an open-source firmware implementation, kernel drivers, and a supportive team at Google. We will see later why the open firmware is important.

The second targetted device was any plain old UVC-compatible webcam. While webcams don't benefit as much from libcamera as ISPs, we wanted to show that the camera stack could also perfectly support what most Linux users are using today.



Acer Chromebook Tab 10



ROCK PI 4

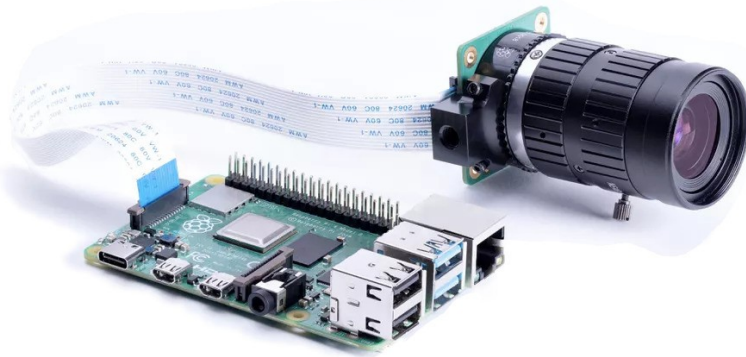


And ARM Too (RK3399)

Shortly afterwards we added one final device to the set, with the ISP found in the Rockchip RK3399. Its architecture is quite different compared to the IPU3, and we wanted to test the libcamera design with different device architectures.

Our initial development device was also a Chromebook, but the same code works on other RK3399 machines, such as the ROCK PI 4 board.

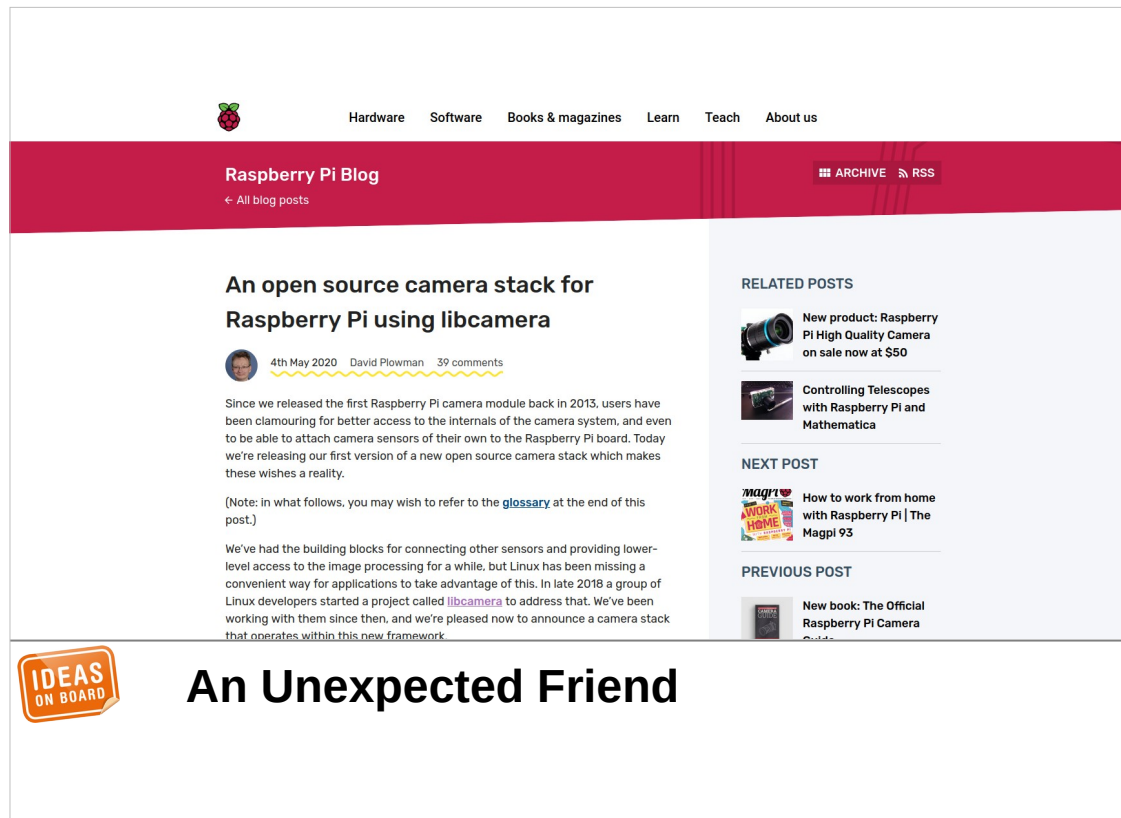
At that point, something unexpected happened.



Raspberry Pi 4 + IMX477

Among the platforms we hadn't considered for libcamera was Raspberry Pi. They released their first camera module in 2013, with a camera stack implemented in a closed-source firmware, out of reach of Linux, and thus, of libcamera.

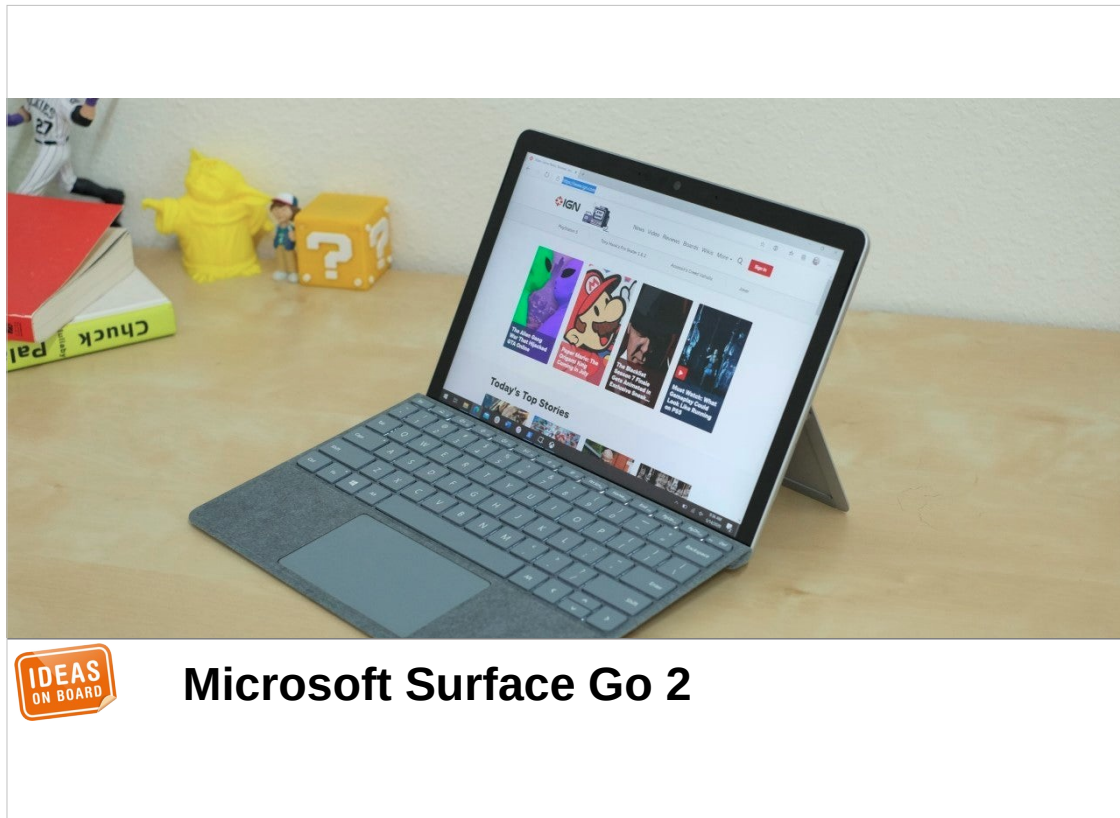
The Raspberry Pi user base wanted better access to the internals of the camera system, and Raspberry Pi listened. They had the code to control the camera pipeline, a complete implementation of image processing algorithms, and a will to open-source it. What was missing was a standard camera stack for Linux that could host those components.



And they got in touch towards the end of 2019. We worked together for about 6 months (which to be fair was mostly them doing the work, with our guidance), and in May 2020 they announced the new camera stack based on libcamera.

The code they released was, as far as I know, the world's first open-source production-quality implementation of image processing algorithms for an ISP. Along with it came a tuning tool, and very detailed documentation.

This made the Raspberry Pi a great platform and playground for image algorithms development. It was a world's first, and even if nobody noticed, we knew it was a major milestone. libcamera had its first experience of collaboration, and its first real users.

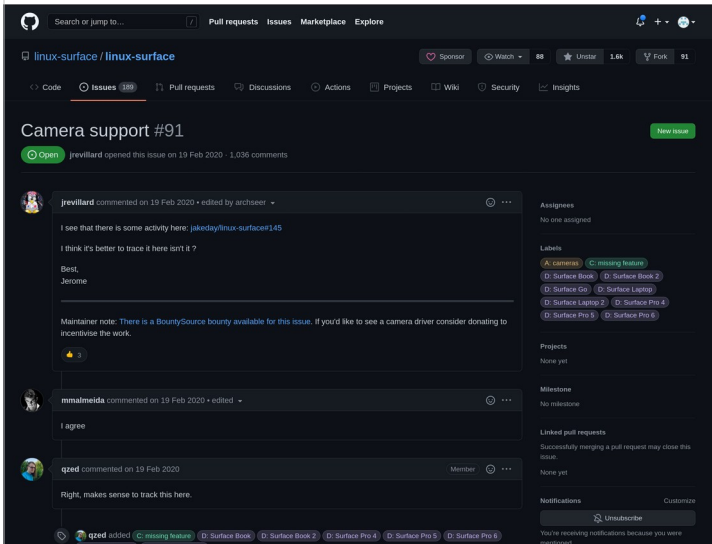


Microsoft Surface Go 2

The second unexpected encounter was, indirectly, due to Microsoft.

We knew about the Windows-based laptops with an IPU3, and we knew they would be challenging to support. The Linux kernel needs to know what camera sensors are present in the system, how they are connected, and it needs to control their power to turn them on and off. On x86 machines, this is normally handled by the ACPI firmware, which describes the hardware, and controls the power automatically. However, on those machines, unlike on Chromebooks, the ACPI description is very badly designed, is missing critical information, and requires drivers to implement power management manually.

Without support from the device manufacturer, without access to the schematics, without firmware or Windows driver source code, there was little that could be done.



Camera support #91

Open jrevillard opened this issue on 19 Feb 2020 · 1,036 comments

jrevillard commented on 19 Feb 2020 • edited by archseer

I see that there is some activity here: [jakeclay/linux-surface#145](#)

I think it's better to track it here isn't it?

Best,
Jerome

Maintainer note: There is a [BountySource](#) bounty available for this issue. If you'd like to see a camera driver consider donating to incentivise the work.

mmalmelida commented on 19 Feb 2020 • edited

I agree

qzed commented on 19 Feb 2020

Right, makes sense to track this here.

qzed added [Missing feature](#) [Surface Book](#) [Surface Book 2](#) [Surface Book 2](#) [Surface Laptop 2](#) [Surface Laptop 3](#) [Surface Pro 4](#) [Surface Pro 5](#) [Surface Pro 6](#)

Labels

- [Camera](#) [Missing feature](#)
- [Surface Book](#) [Surface Book 2](#)
- [Surface Laptop 2](#) [Surface Laptop 3](#)
- [Surface Pro 4](#) [Surface Pro 5](#) [Surface Pro 6](#)

Assignees

No one assigned

Projects

None yet

Milestone

No milestone

Linked pull requests

Successfully merging a pull request may close this issue.

None yet


Notifications

Unsubscribe

You're receiving notifications because you were subscribed.

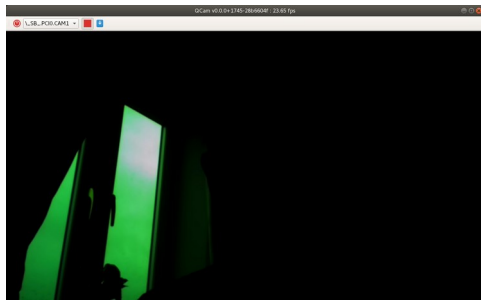
"I found libcamera based on this bug report, it seems to have the required userspace code to have ipu3 working on 5.0."

(archseer)

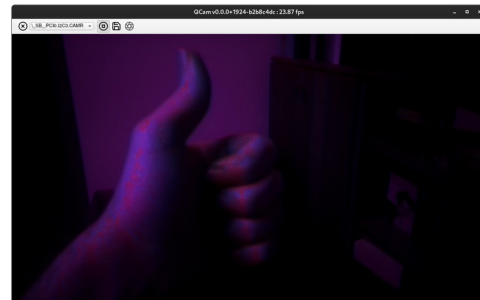
 github.com/linux-surface

That's where the linux-surface community comes into play. They are a set of users-turned-developers who had different Microsoft Surface machines and teamed together to try and get Linux up and running on them.

Needless to say, they were not pleased with lack of camera support. They studied the camera ACPI description and went to reverse engineer the firmware to obtain the missing information. We helped them with that task, which ended up requiring significant effort. After great work on the kernel side from some of the community members, and nearly being driven crazy by ACPI atrocities, they managed to get the sensors detected. libcamera was ready to capture the first images from those devices.



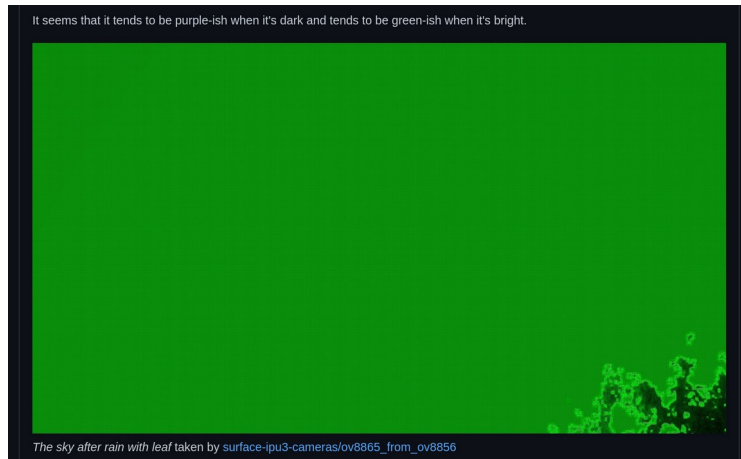
*Initial support status with work
on kernel drivers from djrscally,
kitakar5525 and qzed*



linux-surface

This is how they looked like. You see, at that time, libcamera had support for the IPU3, but no corresponding image processing algorithms. That is how bad it gets without them.

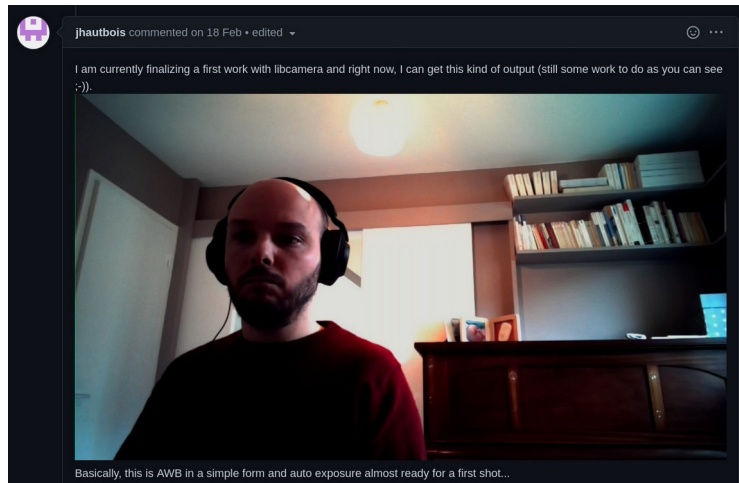
Still, it was a break-through achievement, and I wouldn't have bet on 6 months earlier. Far from being discouraged by the appalling quality, users tried to hack around to improve it. Some of them even got poetic.



Artwork

A user noticed that pictures had a tendency to be purple in low-light conditions and green in bright light conditions. They took a picture through the window on a bright day, and named it “The sky after rain with leaf”. The story doesn’t tell if it was later sold as a non-fungible token.

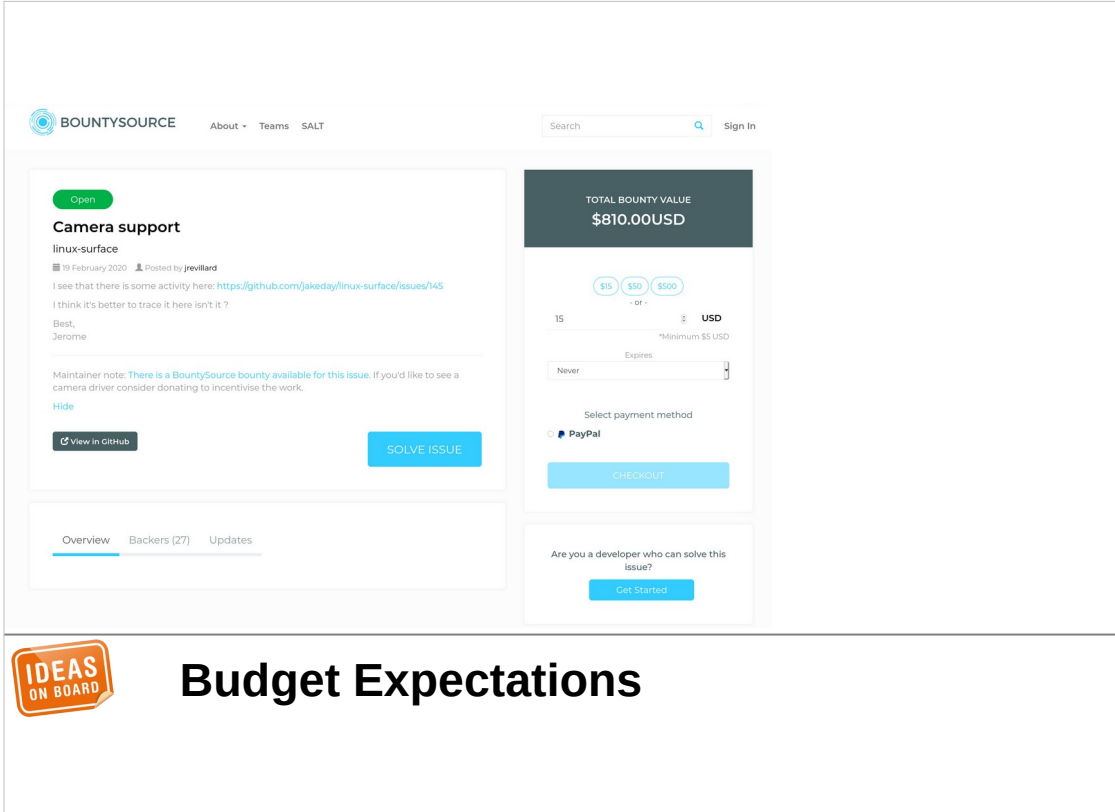
Art is great, but it doesn’t make for a great webcam. So we decided to fix this, and one of our developers spent a few months implementing initial algorithms for the IPU3.



libcamera Involvement

This is what he achieved. As you can see, there's lots of room for improvement, for instance to address lens shading, but the result starts looking like a camera.

At that point, more than two years of work had gone into libcamera. This underlines my earlier point: without a userspace camera framework, the development effort to write a camera application is simply not realistic.



Camera support
linux-surface
19 February 2020 · Posted by [jrevillard](#)
I see that there is some activity here: <https://github.com/jakeday/linux-surface/issues/45>
I think it's better to trace it here isn't it ?
Best,
Jerome

Maintainer note: [There is a Bountysource bounty available for this issue](#). If you'd like to see a camera driver consider donating to incentivise the work.

[View in GitHub](#) [SOLVE ISSUE](#)

Overview Backers (27) Updates

TOTAL BOUNTY VALUE
\$810.00USD

\$15 \$50 \$500
- or -
USD
Minimum \$5 USD

Expires
Never

Select payment method
☒ PayPal

[CHECKOUT](#)

Are you a developer who can solve this issue?
[Get Started](#)

IDEAS ON BOARD

Budget Expectations

Still, this reality isn't widely known among Linux users. This screenshot shows a bounty for the camera support in Linux on Microsoft Surface machines. \$810 to cover all the kernel and userspace development is a bit of an effort underestimation.

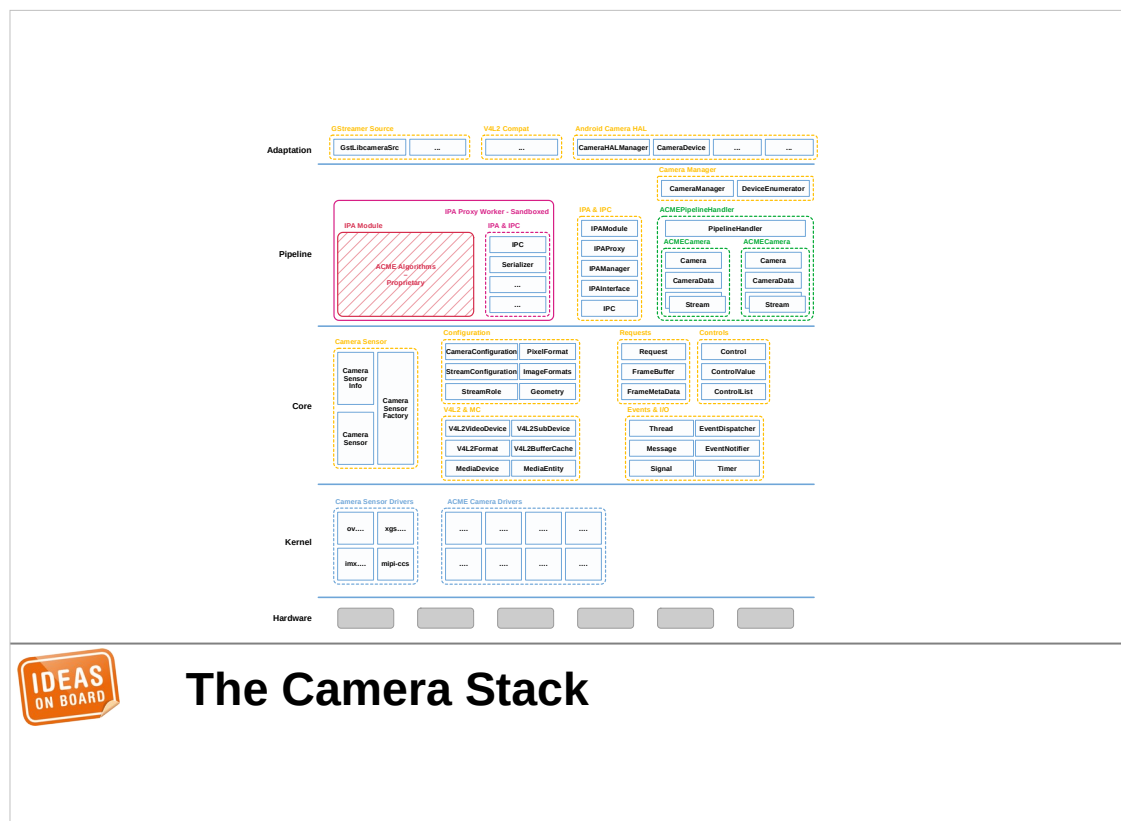
Of course lots of free software gets developed by hobbyists in their free time, and the community has great talents (as seen by the amazing GPU reverse engineering projects). Nonetheless, with the number of different ISPs on the market today, and new ones being developed all the time, solving the issue of camera support on Linux will require involving vendors.

+ - / \ - +
| (o) |
+ - - - - +

libcamera Today

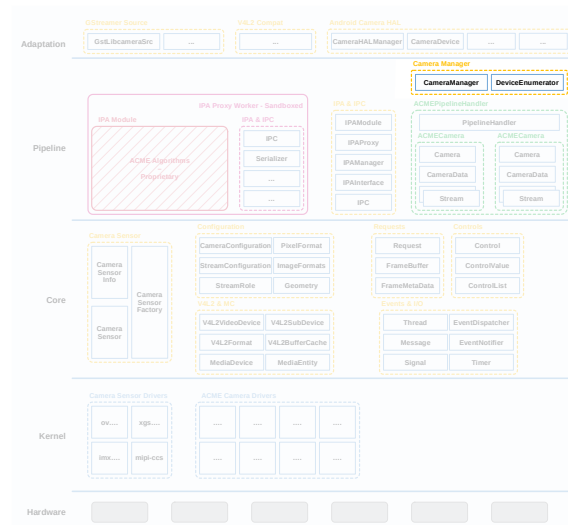


The journey leads us to today. What have we achieved so far with libcamera, and what are we busy working on ?



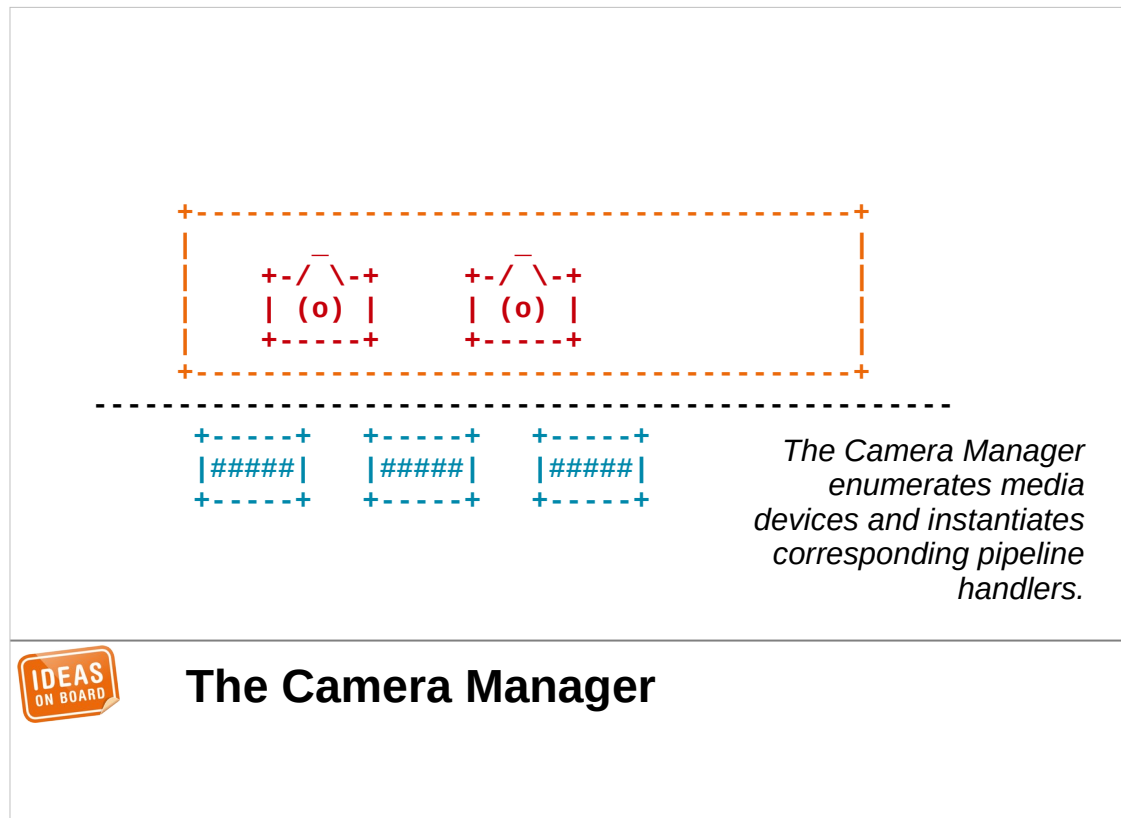
The Camera Stack

I think it's fair to say that we have managed to create a userspace camera stack for Linux.

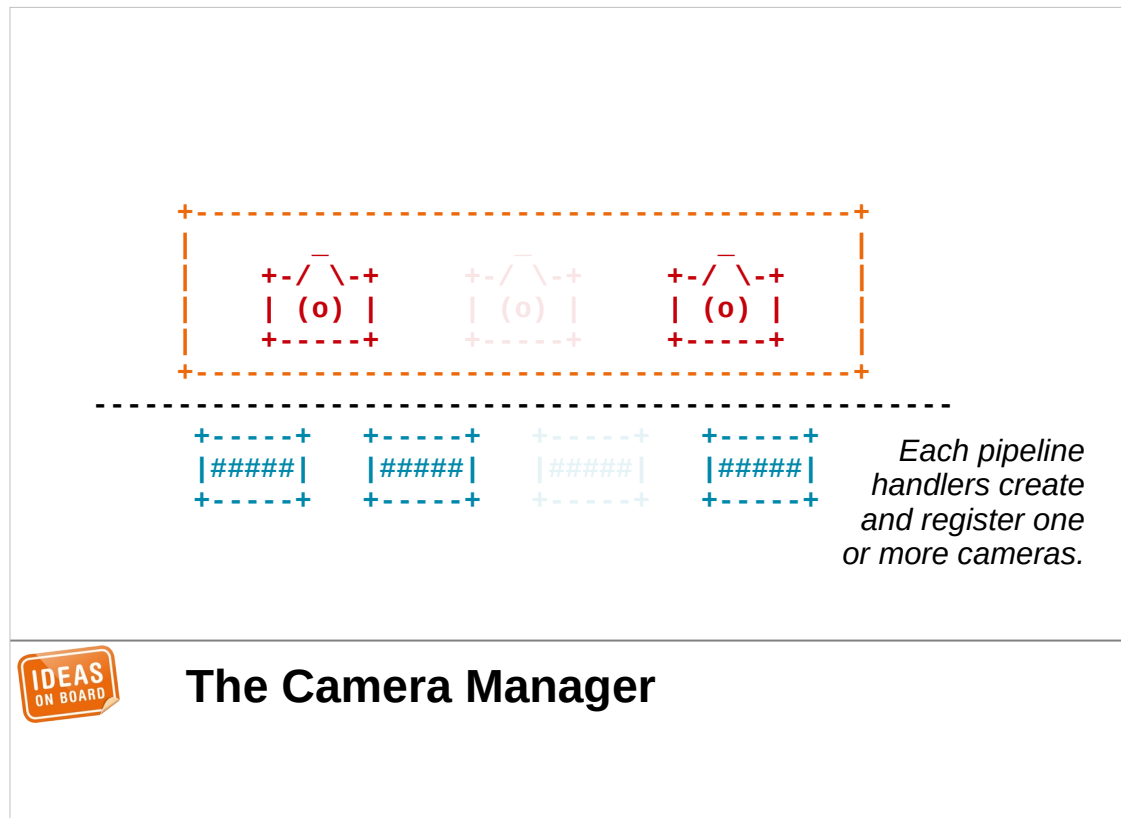


The Camera Manager

As the central piece of the stack, we have a camera manager.

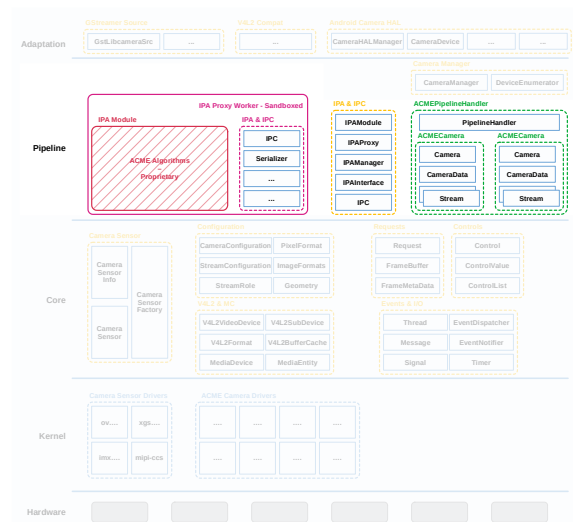


It can enumerate all media devices in the system, and support hotplug to notify applications of camera addition and removal. The camera manager pairs media devices with device-specific backends that we call pipeline handlers.



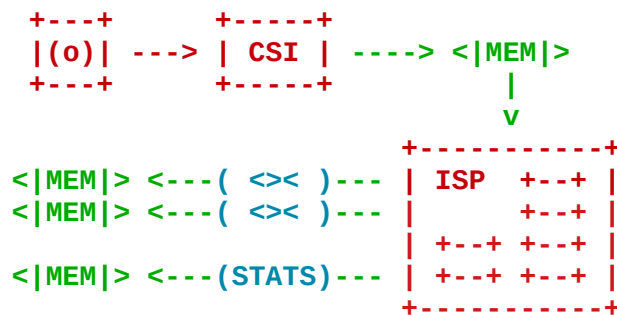
The pipeline handlers in turn create one or more cameras and register them with the camera manager. From that point, the cameras are visible to applications and ready to be used.

So how does it work behind the scenes ?



The Pipeline

As we've seen before, a camera needs a complex pipeline of operations. To support this, the device-specific backend is split in two parts.



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



The Pipeline Handler

First we have the pipeline handler. This is the code that is responsible for all interactions with kernel devices. It configures data routing inside the devices, manages internal memory buffers, handles image capture, ... Overall it abstracts the details of device operation and exposes video streams to applications.

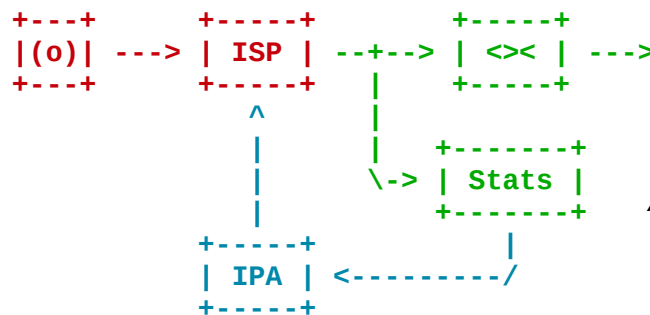
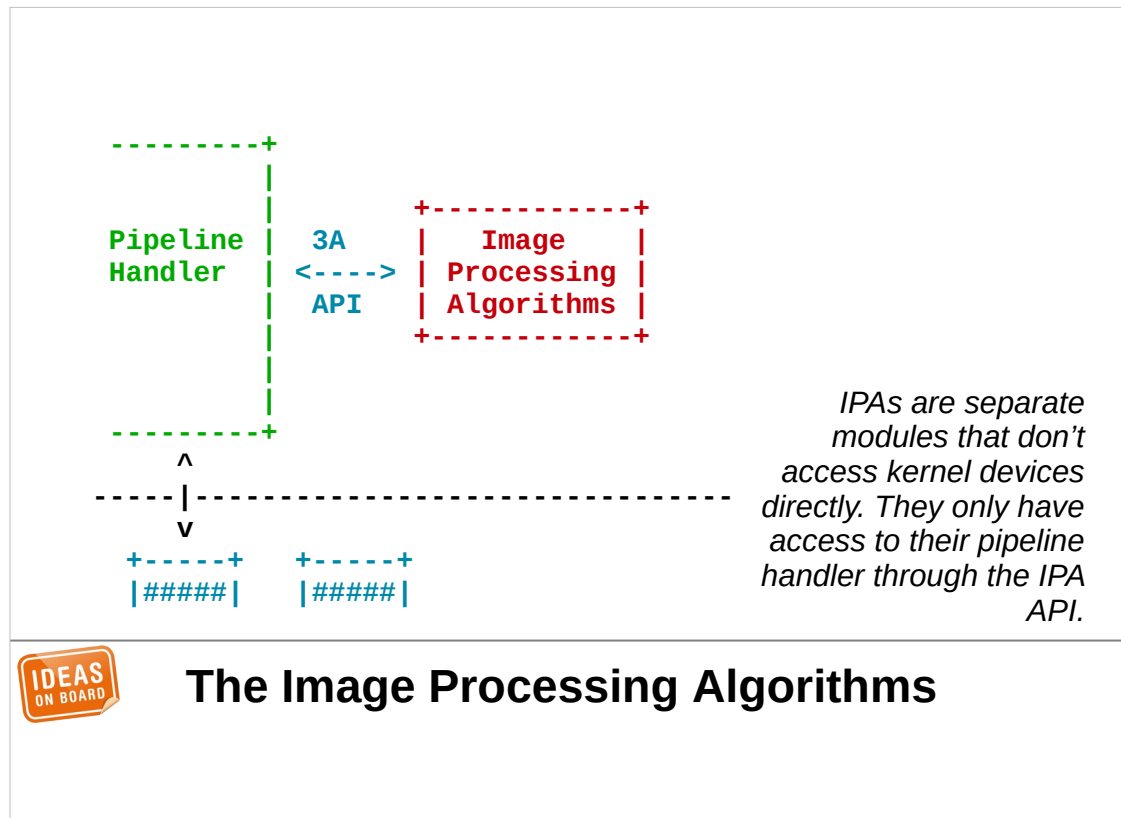


Image Processing Algorithms (IPA) receive statistics from the hardware and compute optimal image parameters.



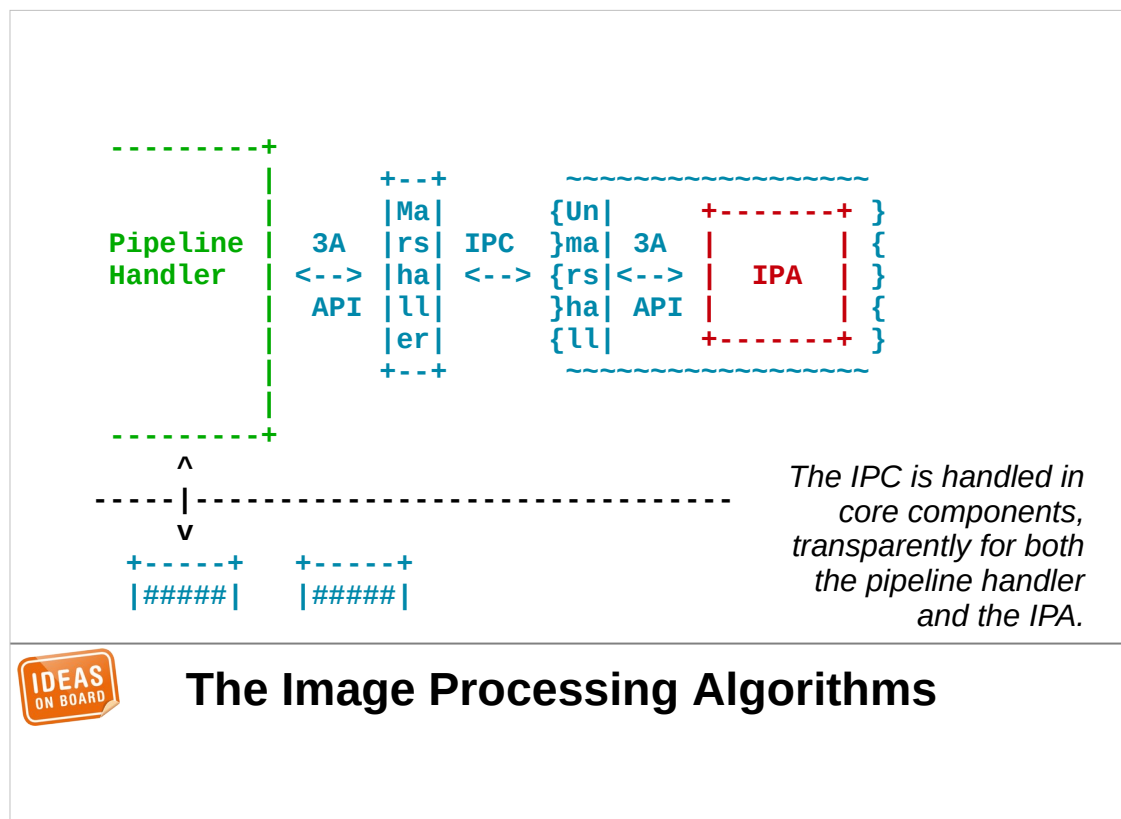
The Image Processing Algorithms

The pipeline handler doesn't implement the image processing algorithms that produce the ISP parameters. This task is performed by the separate IPA module. The module parses statistics produced by the hardware and computes parameters for the sensor and the ISP.

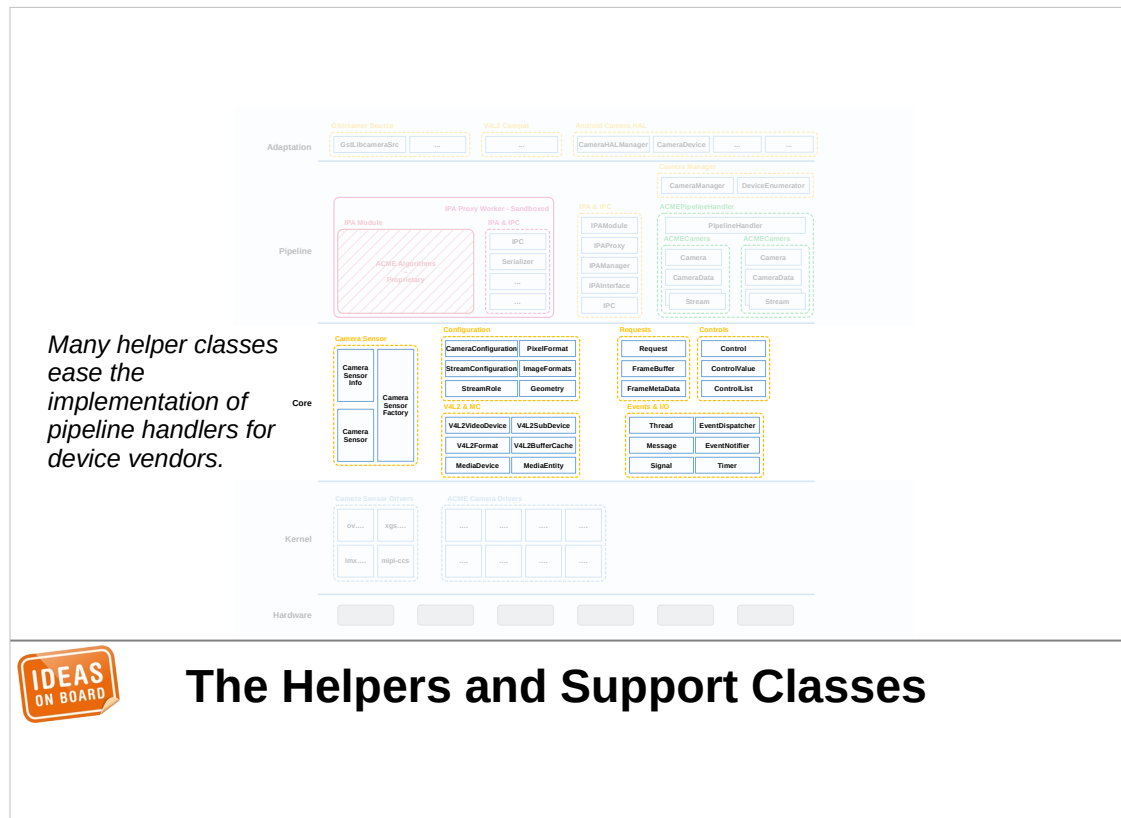


IPA modules don't have direct access to kernel devices. Their only way to communicate with the rest of the system is through the pipeline handler. They receive statistics and other metadata from the pipeline handler, and send the computed parameters back. The parameters will be applied to the devices by the pipeline handler itself. This means that closed-source IPA modules can't cheat and access undocumented devices interfaces when nobody is looking.

The IPA modules are implemented as shared objects that are loaded dynamically. This allows coexistence of IPA modules developed by the libcamera project with third-party modules supplied by camera vendors.

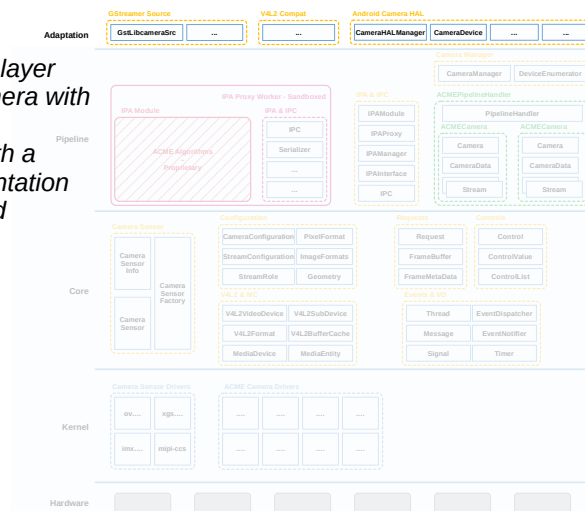


We minimize the impact of sandboxing to keep development simple. The IPC mechanism to communicate with the IPA module process is handled transparently for both the pipeline handler and the IPA module.



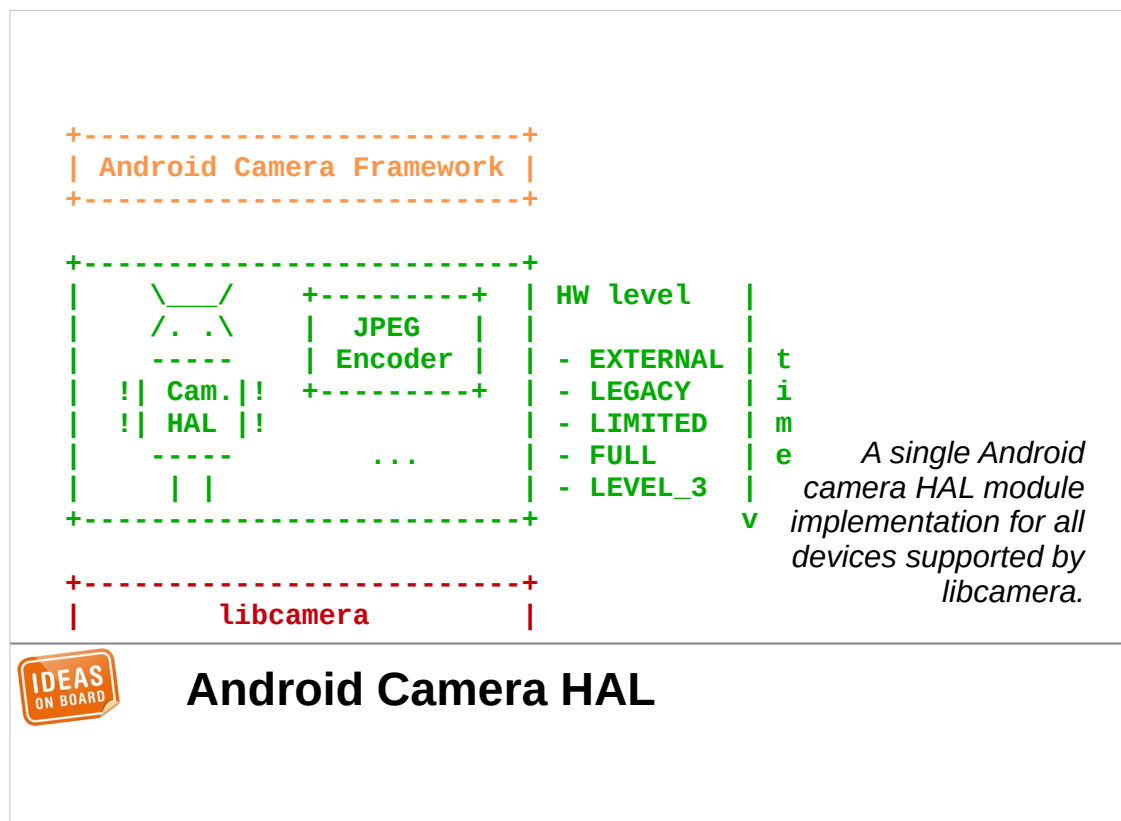
To further ease development, libcamera provides a large set of helper classes. They cover various areas such as wrappers for the V4L2 and MC APIs, pixel format abstraction, threading and message passing helpers, or a logging infrastructure. These are not areas where anyone innovates, but today everybody ends up reinventing the same wheel in their camera stack. With libcamera, vendors can rely on well tested helpers and focus solely on the pipeline handler and IPA modules.

The adaptation layer interface libcamera with other APIs and frameworks, with a single implementation for all supported devices.



The Adaptation Layer

All of this exposes cameras to applications through the libcamera native API, but libcamera offers more. The adaptation layer sits on top of the native API, and interfaces libcamera with other APIs and frameworks. Because all device-specific code is located in the pipeline handler and IPA module, the adaptation layer is implemented once and works on all devices.



In no particular order, the first component in the adaptation layer is an Android camera HAL module. This implements a camera provider for the Android camera service, and gives free Android support for any platform supported by libcamera.

And as mentioned earlier, because Chrome OS uses the same camera HAL module API as Android, libcamera also supports Chrome OS.

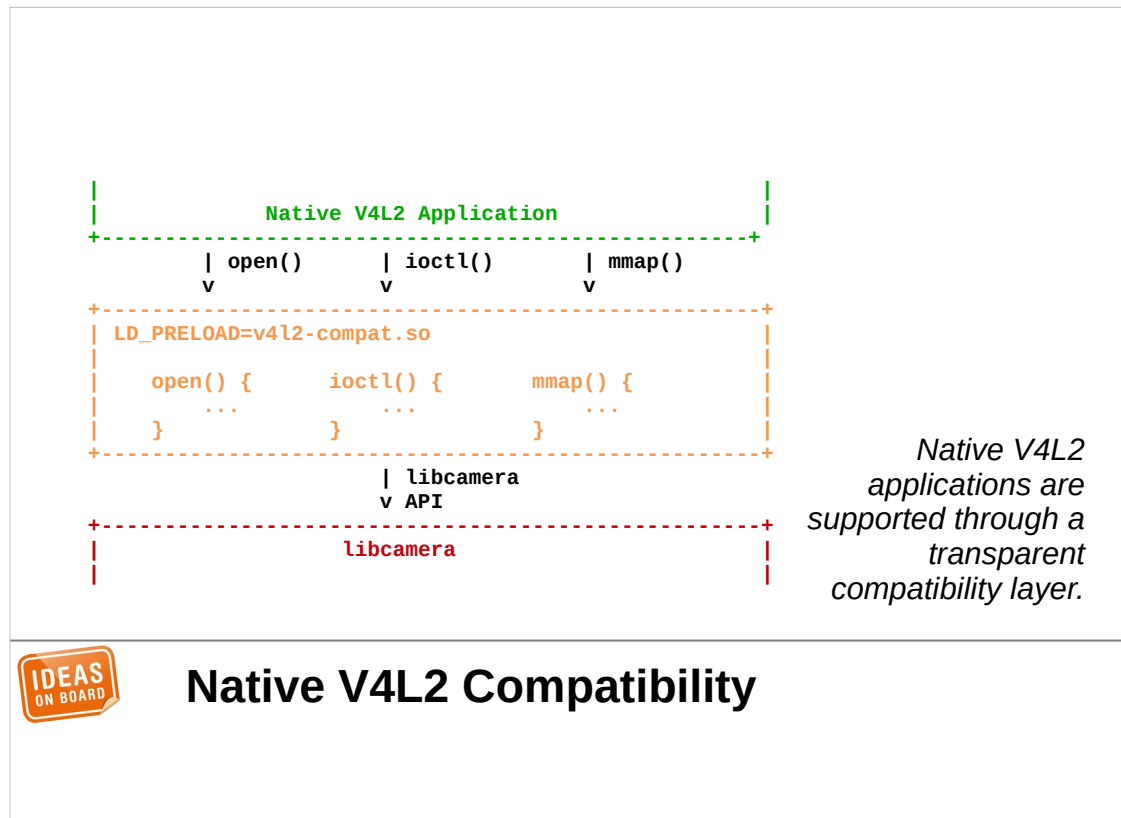


*"libcamerasrc" offers a
multi-stream source
element for GStreamer
applications.*



GStreamer

The second adaptation component is a GStreamer source element named libcamerasrc. It provides multi-stream capture in a GStreamer pipeline for any camera supported by libcamera. It has been successfully tested with Cheese, the GNOME camera application.



The last adaptation component implements V4L2 emulation. It allows existing V4L2 applications to use libcamera without any modification, without even being recompiled. We achieve this with a shared object that is preloaded inside the application and intercepts calls to the C library. When it detects that those calls attempt to access a V4L2 device, it redirects them to the V4L2 emulation code that translates the calls to the libcamera API.

This is a best effort approach, as it's impossible to fully emulate 100% of V4L2. We have however successfully made video calls with Firefox using this technique.

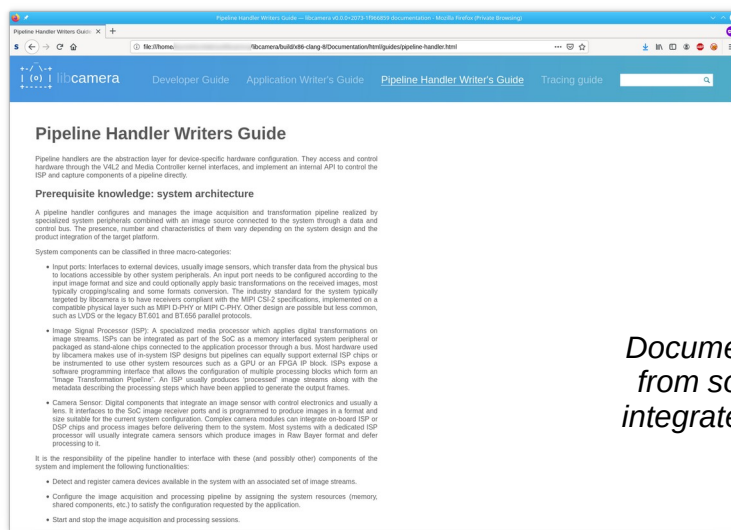
Features	Status
Core	Multi-camera, multi-stream, per-frame control, hotplug
Supported platforms	Raspberry Pi 3&4, Intel IPU3 (Kaby Lake), Rockchip RK3399, UVC, NXP i.MX7, Allwinner A31
IPA modules	Raspberry Pi, Intel IPU3 IPA module isolation with IPC (based on mojom IDL)
Adaptation layers	GStreamer source element (with multi-stream), Android camera HAL v3.3, V4L2 emulation
Tooling	Camera Tuning Tool (Raspberry Pi), tracing infrastructure and analysis script
Applications	cam (command line Swiss army knife), qcam (GUI), simple-cam (tutorial)
Documentation	Extensive API documentation and high-level tutorials and guides available



Summary of Supported Features

Let's summarize the currently supported features in one slide. We have a camera stack, it supports multiple cameras, with multiple streams per camera, per-frame control and hotplug. The list of supported devices is limited but growing, with our flagship implementation being on Raspberry Pi today. Raspberry Pi and IPU3 both have an IPA module. The last two devices in the list have no ISP, so only smart sensors can be used.

We've just seen the three components of the adaptation layer, with GStreamer, Android and V4L2 being supported. libcamera also includes tools, such as a camera tuning tool for Raspberry Pi, and a tracing infrastructure for debugging. We have three sample applications, one command-line application that implements all supported features, one GUI application, and one simple application used as a tutorial. We have extensive documentation of the API with 100% coverage, as well as high-level tutorials and guides.




Documentation compiled from source tree, will be integrated in the website.



Guides

The API documentation is published on the libcamera website with nightly builds. The guides are currently available from the source repository only, they can be compiled to HTML, and we will publish them on the website too.

Many developers are better at coding than writing documentation, when they don't viscerally hate it. Documentation takes time, it's an investment, but one of the lessons that libcamera taught me is that it pays off to enforce a good documentation policy from day one.



The image shows a blue printed circuit board (PCB) with various electronic components, including a large central chip, several smaller chips, and various connectors. The board is set against a background of white and light gray geometric shapes.





MEDIATEK | OLogic

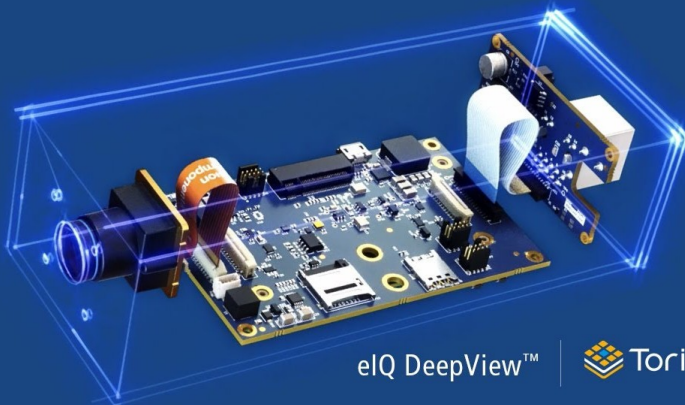
MediaTek Pumpkin i500 (MT8385)

IDEAS ON BOARD


New Platforms

We are of course not sitting idle, and we're working on support for additional platforms. This includes an SoC from MediaTek, used in many IoT applications. The ISP is left out for the time being, so only smart sensors can be used.






elQ DeepView™

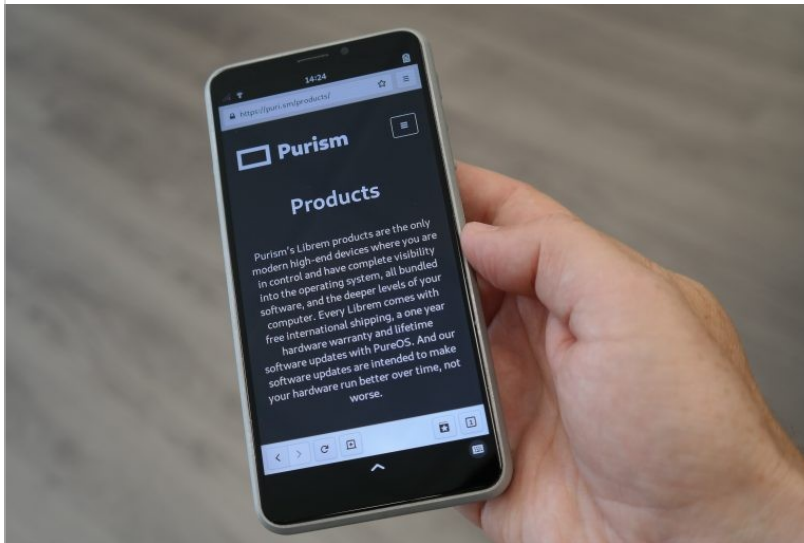
 Torizon™

NXP i.MX8M Plus



New Platforms

We have also started implementing support for the NXP i.MX8M Plus. This is the first SoC from NXP that includes an ISP, and here we're targeting ISP support already.



Purism Librem 5



New Platforms

Another platform that we've started looking at is the Librem 5 phone. It is based on raw sensors but doesn't have a hardware ISP, so we're exploring implementation of a software ISP running on the GPU.

Features	Status
New platforms	MediaTek MT8385 (with YUV sensors), NXP i.MX8M Plus, Librem 5
Open-source IPA modules	Intel IPU3, Rockchip ISP, I.MX8M Plus
Reprocessing API	Work in progress in the libcamera core, Android HAL support will follow
Controls and properties	New controls and properties are continuously added on a per-need basis
API cleanups	Moving toward the API freeze for a 1.0 release, API changes will remain backward-compatible (extensible API design, d-pointer design pattern, ...)
Language bindings	Python bindings in progress
Android LIMITED and FULL CTS compliance	Core infrastructure ready, controls and properties (static, control and dynamic metadata) being developed incrementally
Integration	Native support in Chromium web browser available at https://github.com/libcamera-org/chromium .



Work In Progress

There's more work in progress. We're improving IPA modules for the Intel IPU3 and Rockchip ISP, with the goal to bring the quality on par with Raspberry Pi. We are also extending the libcamera API with new features and new use cases, and at the same time we're cleaning the API to move towards a first 1.0 release.

There's lots of work in progress on the Android camera HAL implementation to pass the Android conformance test suite, and we're also implementing Python bindings for libcamera to support new communities of users.



Chromium (on MS Surface Go 2)

On the integration side, we have a prototype of native libcamera support for the Chromium web browser. This is a screenshot of a video call in Chromium, with Jean-Michel on the top right using a Surface Go 2 and libcamera.

This is running an old version of libcamera, from before we had IPU3 algorithms. Today the quality is much better, and you wouldn't notice libcamera is involved. This is likely how we'll judge libcamera's success in the end, we'll have done a good job if users don't notice we exist.

Embedded Camera

Exploratory Group for Embedded Camera and Sensors



emva
european machine vision association

HOSTED BY:

KHRONOS
GROUP



Participation In Industry Initiatives

Another part of our work that is not very visible is our participation in industry initiatives. We are an active member of the Embedded Camera exploratory group hosted by Khronos and the European Machine Vision Association. The goal of the group is to explore opportunities to standardize a camera API. Quite obviously we think that libcamera is the right solution.

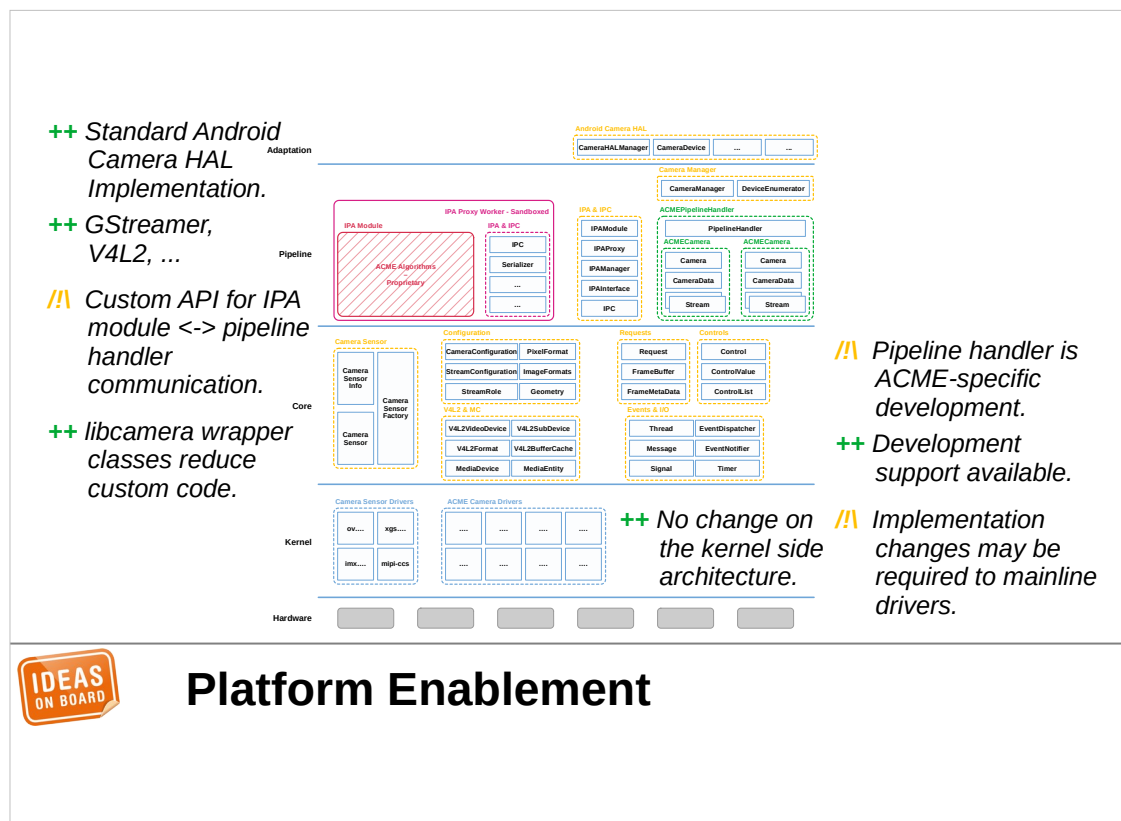
In parallel, we also have multiple bilateral contacts with SoC vendors to work on libcamera adoption.

+ - / \ - +
| (o) |
+ - - - - +

For
Camera
Vendors



Speaking of vendors, we've already seen some of the advantages that libcamera brings, let's now have a look at what implementing the libcamera stack entails



At the bottom of the stack, we have the kernel drivers. As long as they implement the MC API, no change is required on the kernel side to work with libcamera. This being said, upstreaming kernel drivers may of course requires changes as part of the review process with the kernel community.

*We drive MC and V4L2
standardization and
extensions development
according to our needs.*



Kernel APIs

libcamera has driven the development of extensions in V4L2 and MC to fulfil the needs of the platforms we work with. We have also encountered ambiguities and design deficiencies in V4L2 and worked on fixing them. The libcamera team has extensive experience with kernel development in the media subsystem, so we can also help vendors in this area if their platforms have needs that are not covered yet.

On a side note, it was an interesting experience of humility to realize that some of those problems in V4L2 were actually in APIs that I had designed myself. That's another lesson learnt from libcamera, a kernel API that only gets validated with test tools, without a real userspace stack, will most likely have defects.

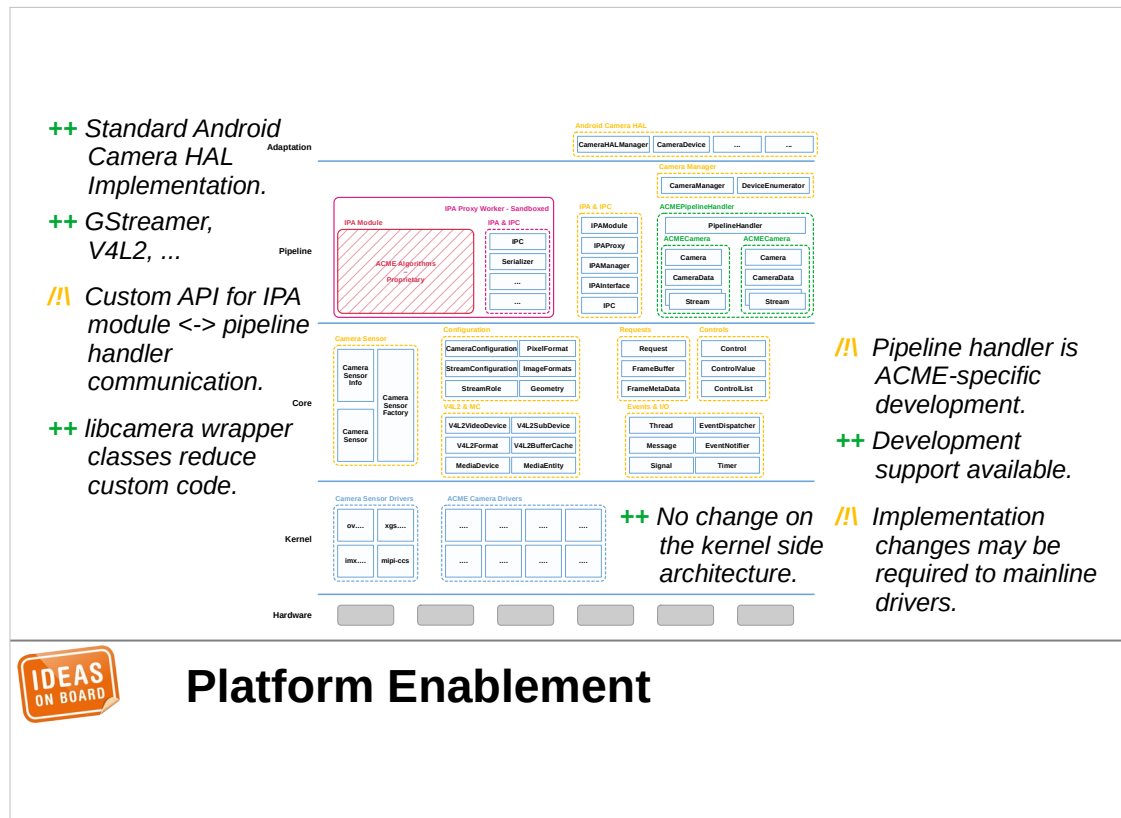
*We drive MC and V4L2
standardization and
extensions development
according to our needs.*

*libcamera is a
userspace framework,
not a hostile takeover
of kernel development.*



Kernel APIs

libcamera is however a userspace framework, not a hostile takeover of kernel development, so we can't help vendors to bypass the requirements of the kernel community.



We have already seen that libcamera provides an extensive set of helpers that help reducing development complexity and time by avoiding the need to reinvent the wheel. The adaptation layer is also shared by all cameras, freeing vendors from having to write Android or GStreamer support manually.

The only components that need to be developed specifically for a platform are the pipeline handler and IPA module. This is the responsibility of the vendor. There is extensive documentation, examples pipeline handlers are available to provide guidance, and we are also here to provide support.

*The libcamera core
is licensed under the
LGPL v2.1 or later.*



Licensing

A word on licensing, because it's important. The libcamera core and the adaptation layer are licensed under the LGPL.

*The libcamera core
is licensed under the
LGPL v2.1 or later.*

*Changes need to be published
according to the license. This
includes pipeline handlers.*



Licensing

This includes the pipeline handlers, which need to be published according to the license. The IPA modules are excluded.

*The libcamera core
is licensed under the
LGPL v2.1 or later.*

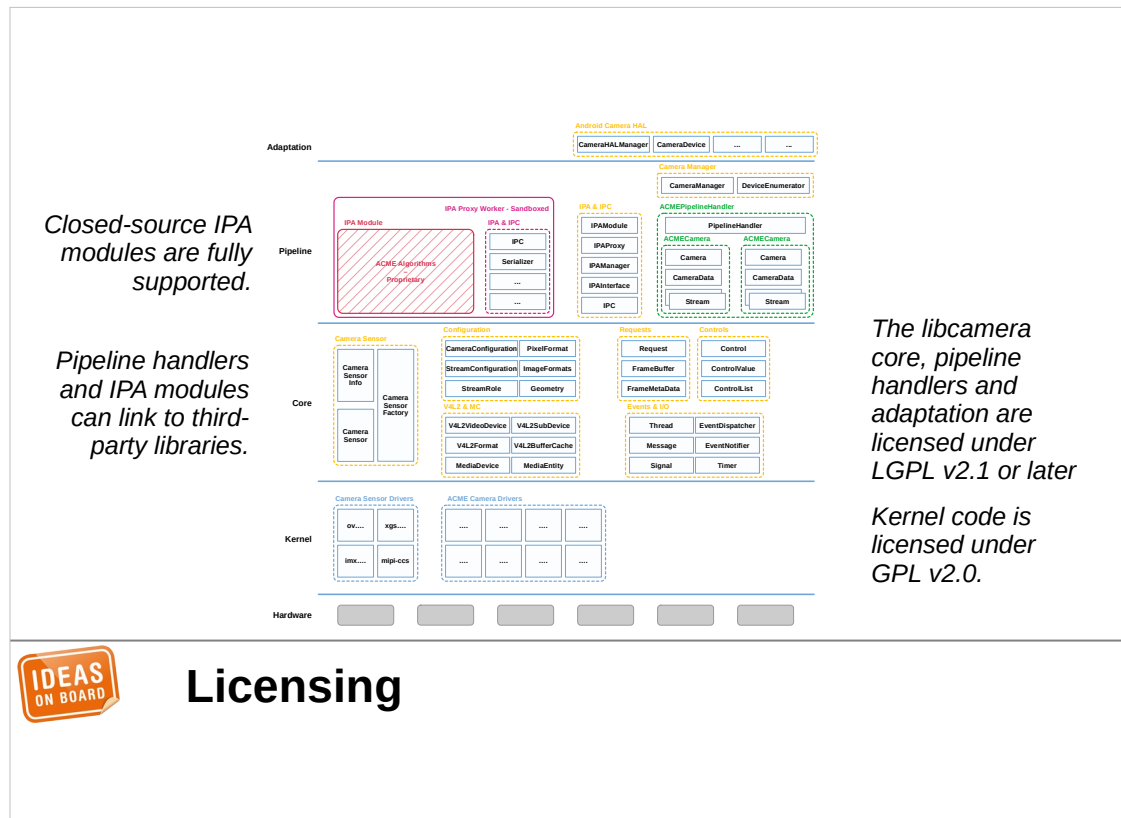
*Changes need to be published
according to the license. This
includes pipeline handlers.*

*Upstreaming is not mandatory
but highly recommended
(forks are costly to maintain).*



Licensing

Only publishing the code is required to comply with the LGPL, upstreaming it isn't a requirement. We however strongly recommend upstreaming. The best results are achieved by working together, and forks are costly to maintain.



The kernel code is of course covered by its own license, which is out of scope for libcamera. Let's also note that both the pipeline handlers and IPA modules can link to third-party libraries if desired, as long as the licenses are compatible. Closed-source IPA modules are fully supported as discussed before, even if we would like to encourage vendors to follow the lead of Raspberry Pi and open the algorithms too.

+ - / \ - +
| (o) |
+ - - - - +

An Exciting Future



This presentation would of course not be complete without talking about the future of libcamera. I'll present here features that we envision but haven't started developing yet.

Features	Status
Per-stream controls	Concept approved, will be scheduled in the future.
Zero shutter lag	Will be possible through the reprocessing API. We are considering a high-level “use cases” library on top of libcamera for ZSL and similar features.
Exposure bracketing HDR	Similarly to ZSL, could be implemented in a “use cases” library. A solution for device-assisted HDR (hardware merging or software merging based on hardware-generated metadata) is needed.
Logical camera devices (W+T zoom, power saving, ...)	Not planned yet, missing development and test platform.
Still image trigger sequence (focus & flash)	Not planned yet, missing development and test platform.



Future Features – Core

In the libcamera core, we have already thought about quite a few interesting features. We have per-frame controls, but they’re currently global to the camera, and we want the ability to set per-stream controls too. This could allow setting for instance different digital zoom factors for different streams.

We are also considering a higher-level “use cases” library on top of libcamera, to offer features such as zero shutter lag capture or exposure bracketing HDR. Both of these would be implemented by capturing raw frames, pre-processing them in software, and sending them back to the ISP for further processing.

Two other important features that will eventually make their way in libcamera are still image triggering with focus and flash support, and logical cameras. Logical cameras is something available in many phones today. It’s the ability to combine multiple physical sensors to create one logical camera. There is a variety of use cases, such as seamlessly switching between a wide angle lens and a tele lens when zooming, or using multiple sensors to infer depth information.

Features	Status
Open-source IPA modules	Cross-platform core library, long term work to convince device vendors
GPU-based processing	Proof of concept shader code in qcam test application, should be leveraged to create GPU-based ISP for platforms without a hardware ISP (Librem 5).
New devices support	Ongoing discussions with SoC/system vendors, community-driven effort on legacy devices (any volunteer for the N900/N9 ?)



Future Features – Devices

We really want to expand the number of supported devices. I've already mentioned ongoing work with the Librem 5 and a GPU-based ISP implementation, and we have ongoing discussions with more vendors. Support for older devices would also be great, I would personally be very very happy to see the OMAP3 ISP supported in libcamera, and finally bring an open camera stack to the Nokia N900 and N9 phones. That would be a great community project I would love to mentor, so please volunteer, don't be shy.

Wednesday, September 29 • 3:55pm - 4:45pm


Advanced Camera Support on Allwinner SoCs with Mainline Linux - Paul Kociałkowski, Bootlin

Sign up or log in to save this to your schedule, view media, leave feedback and see who's attending!

<https://sched.co/IALq> [Tweet](#) [Share](#)

Capturing pixels with a camera involves a number of steps, from the ADC reading the photosites in the image sensor to the final pixel values that are ready for encode/display, with various processing and transmission taking place along the way. While simple cases put most of the heavy lifting on the image sensor's side (through its embedded processor) and use a simple parallel bus for transmission, advanced cases require more work to be done outside of the sensor. In addition, modern high-speed transmission buses also bring in more complexity. This talk will present how support for such an advanced use case was integrated into the mainline Linux kernel, using the Media and V4L2 APIs. It involves supporting a sensor using the raw Bayer RGB format, transmission over the MIPI CSI-2 bus as well as support for the Image Signal Processor (ISP) found on Allwinner platforms. A specific focus will be set on this ISP, with details about the features it implements as well as the internal and userspace APIs that are used to support it. The integration between all of the involved components will also be highlighted.

Speakers



Paul Kociałkowski
Embedded Linux Engineer, Bootlin
Paul joined Bootlin in 2018 and started with bringing support for the Allwinner VPU to mainline Linux. He went on to cover more topics related to graphics and multimedia, with various contributions to the DRM and V4L2 Linux subsystems as well as various related projects. Before that... [Read More](#) --


Wednesday, September 29, 2021 3:55pm - 4:45pm PDT

Room 4

Embedded Linux Conference (ELC), Streaming Media & Graphics

Experience Level Mid-level

Talk Type Virtual



New Platforms

Paul Kociałkowski from Bootlin will talk tomorrow about his work on a kernel driver for an ISP found in Allwinner SoCs. He has posted the code a few weeks ago for review, and it would be an interesting platform to support in libcamera with a community-based effort. If you attend his presentation, don't hesitate to tell him you want to volunteer.


Features	Status
HAL v3.5(+)	On the roadmap, on hold due to lack of development and test platform. Future Android camera HAL API extensions will be implemented (including extensions to the libcamera core if needed).
Zero Shutter Lag	HAL-based ZSL implemented using the libcamera reprocessing API.



Future Features – Android HAL

We will of course continue working on the Android camera HAL and support newer versions of the HAL API. Support for additional features, such as zero shutter lag, is also in scope.

Features	Status
Frameworks	PipeWire, OpenCV, Qt Multimedia, Electron, <insert your framework here>, ...
Applications	Firefox, OBS, <insert your application here>, ...
Operating Systems	Chrome OS, Android, Linux distributions, Buildroot, OpenEmbedded, ...



Future Features – Integration

libcamera wouldn't be very useful if it wasn't integrated in frameworks, applications and distributions. In the frameworks category, PipeWire is particularly in scope, but so are OpenCV, Qt Multimedia, Electron or your favourite framework.


On the application side, while Firefox already works with libcamera using V4L2 emulation, native support would be better. I was also very tempted to add native libcamera support to OBS when recording this presentation, and I hope that many other applications will follow.

On the operating system side, we're packaged by Chrome OS and buildroot already, by Debian unstable too, and work is ongoing for Fedora. This will take more time, but being included in Android AOSP is something I'm looking forward to.

+ - / \ - +
| (o) | libcamera
+ - - - - +



This concludes the presentation. The slides should now be available from the Linux Foundation website. I hope you found this interesting, and regardless of whether a user, an application developer or a vendor, please don't hesitate to come and talk to us.



libcamera-devel@lists.libcamera.org
<irc://chat.freenode.net/#libcamera>

laurent.pinchart@ideasonboard.com



Contact

The libcamera team can be contacted through our public mailing list and IRC channel, and you can also contact me directly by e-mail. I am now available for questions in the conference chat channel.

Thank you.



Thank you for attending, and I hope you will enjoy the rest of the conference.