

Camera HAL



Android's camera hardware abstraction layer (HAL) connects the higher-level camera framework APIs in Camera 2 to your underlying camera driver and hardware. The camera subsystem includes implementations for camera pipeline components while the camera HAL provides interfaces for use in implementing your version of these components.

You're absolutely right, and here's a clearer breakdown of what that means:

Android Camera HAL: Explained Simply

1. What is the Camera HAL?

The **Camera Hardware Abstraction Layer (HAL)** is a critical layer in Android that connects:

- **High-level APIs** like `android.hardware.camera2` (used by apps)
→ to
- **Low-level hardware drivers** (the actual camera sensor and ISP control code)

It allows Android to work across many different camera hardware designs by providing a **standardized interface**.

2. Why is Camera HAL Needed?

Android runs on many devices with very different camera hardware. The HAL acts like a **translator**:

- The **Camera2 API** issues requests like "capture a frame with 1/100s shutter, ISO 200, and focus at 3 meters".
- The **Camera HAL** takes those requests and tells the **camera driver/hardware** how to fulfill them.
- The HAL also sends back the **captured image and metadata**.

3. What Does the Camera HAL Do?

The HAL:

- Initializes the camera
- Reports camera capabilities to the framework
- Handles request/result queues
- Controls frame rate, autofocus, flash, exposure, white balance

- Delivers raw or processed frames to the framework

In modern devices, the HAL must implement **multiple cameras, stream configurations, buffer management**, and **hardware pipelines (e.g., ISP)**.

4. HAL Versions and Evolution

- **Camera HAL v1** → Simple, used with the old `Camera API` (deprecated).
- **Camera HAL v3+** → Advanced, supports `Camera2 API` with full control over the image capture pipeline.

If your HAL supports **Camera HAL v3**, then your device can expose **manual controls, RAW capture, and advanced streaming** features.

5. HAL Interface Implementation

Vendors implement the HAL by writing C++ code that matches the camera HAL interface defined in AOSP:

- The entry point is `camera_module_t (camera_module)`
- Key interfaces include:
 - `camera_device_t` (open/close, stream config)
 - `camera3_device` (for HAL3)
 - `process_capture_request()` and `notify()` callbacks

These implementations are compiled into a `.so` (shared object) and placed in `/vendor/lib(64)/hw/`.

6. Relationship with Other Components

- **Camera HAL** interacts with:
 - **Camera driver** in kernel space (via `/dev/video*`)
 - **ISP firmware** or secure pipelines (often via TEE or vendor blobs)
 - **Gralloc** for buffer allocation
 - **SurfaceFlinger** or app buffers for preview

Let me know if you'd like a sample HAL code structure or flow diagram!



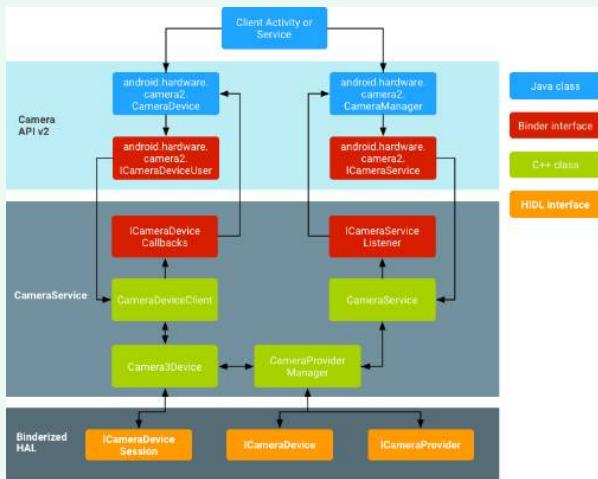


Figure 1. Camera architecture

app framework

At the app framework level is the app's code, which uses the Camera 2 API to interact with the camera hardware. Internally, this code calls corresponding Binder interfaces to access the native code that interacts with the camera.

AIDL

The binder interface associated with CameraService can be found at frameworks/av/camera/aidl/android/hardware. The generated code calls the lower level native code to obtain access to the physical camera and returns data that is used to create the CameraDevice and eventually CameraCaptureSession objects at the framework level.

native framework

This framework residing in frameworks/av/ provides a native equivalent to the CameraDevice and CameraCaptureSession classes. See also NDK camera2 reference.

binder IPC interface

The IPC binder interface facilitates communication over process boundaries. There are several camera binder classes located in the frameworks/av/camera/camera/aidl/android/hardware directory that call into camera service. ICameraService is the interface to the camera service; ICameraDeviceUser is the interface to a specific opened camera device; and ICameraServiceListener and ICameraDeviceCallbacks are the respective CameraService and CameraDevice callbacks to the app framework.

camera service

The camera service, located in frameworks/av/services/camera/libcameraservice/CameraService.cpp, is the actual code that interacts with the HAL.

HAL

The hardware abstraction layer defines the standard interface that the camera service calls into and that you must implement to have your camera hardware function correctly.

Implement the HAL

The HAL sits between the camera driver and the higher-level Android framework and defines an interface that you must implement so apps can correctly operate the camera hardware. The HIDL interfaces for the Camera HAL are defined in hardware/interfaces/camera.

A typical binderized HAL must implement the following HIDL interfaces:

ICameraProvider: For enumerating individual devices and managing their status.

ICameraDevice: The camera device interface.

ICameraDeviceSession: The active camera device session interface.

Reference HIDL implementations are available for CameraProvider.cpp, CameraDevice.cpp, and CameraDeviceSession.cpp. The implementation wraps old HALs that still use the legacy API.

Starting with Android 8.0, Camera HAL implementations must use the HIDL API; use of the legacy interface isn't supported.

Input validation

Because the HAL has access to different resources than the camera service, the boundary between the two is treated as a security boundary. This means parameters passed from the camera service are considered untrusted and unsanitized. To prevent security vulnerabilities that allow attackers to escalate privileges or access data they aren't intended to have access to, the camera HAL must validate parameters passed from the camera service to the HAL. This includes checking that buffer length values are within allowable ranges and sanitizing the parameters before use and before passing them to hardware or kernel drivers.

Thanks for sharing the full architecture and the image (`ape_fwk_camera2.png`). Based on what you've posted and the diagram, let me break down **how everything works end-to-end** in simple steps, connecting the image and your description:

Android Camera Framework Flow (Camera2 API v2) — Simplified Breakdown

1. App Layer (Java Framework)

Topmost level (in blue in the image):

- Your app uses **CameraManager** and **CameraDevice** classes from `android.hardware.camera2`.
- When you open a camera, it communicates via **Binder IPC** with system services using interfaces like:
 - **ICameraService**
 - **ICameraDeviceUser**

2. Binder IPC Layer

- **Binder interfaces** (in red boxes) are the communication glue:
 - **ICameraService**: Used to list/open camera devices.
 - **ICameraDeviceUser**: Controls a specific camera (start/stop preview, capture frames, etc.).
 - **ICameraServiceListener**: Listens for events (e.g., camera connected/disconnected).

- **ICameraDeviceCallbacks**: Sends results/events back to the app.

These are defined using **aidl** (Android Interface Definition Language) and implemented in native code.

📁 Found in: `frameworks/av/camera/aidl/android/hardware`

3. Native Camera Framework (CameraService)

- `CameraService.cpp` in `frameworks/av/services/camera/libcameraservice/` is the main native component.
- `CameraProviderManager` talks to HAL and manages all devices.
- `CameraDeviceClient` is created per opened camera device and talks to:
 - `Camera3Device` (for HALv3 implementation)
- These are C++ classes (green boxes in the diagram).

4. Camera HAL (Binderized HAL using HIDL)

- Bottommost layer (orange boxes):
 - `ICameraProvider` (lists available devices)
 - `ICameraDevice` (represents one hardware camera)
 - `ICameraDeviceSession` (active streaming/capture session)

📁 Located in: `hardware/interfaces/camera/` (HIDL definitions)

From Android 8.0 onwards, **Camera HAL must use HIDL**.

These components are implemented by the **Soc vendor** and loaded from `/vendor`.

5. Camera Driver (Kernel) — Not shown in the image

- The HAL calls into the actual camera driver (usually a `v4l2` device like `/dev/video*`).
- This interacts with the **Image Signal Processor (ISP)**, sensors, and camera hardware.

🔒 Security: Input Validation

- The boundary between `CameraService` and HAL is a **security boundary**.
- HAL must **validate everything** passed from `CameraService` to avoid bugs and privilege escalation:
 - Buffer sizes
 - Input formats

- Metadata values

📌 Real Flow Summary (example):

1. App calls `CameraManager.openCamera()`
2. Goes to `ICameraService.openCamera()`
3. Service creates a `CameraDeviceClient` and `Camera3Device`
4. Uses `CameraProviderManager` → calls `ICameraProvider::getCameraDevice()`
5. Session is created via `ICameraDevice::open() → ICameraDeviceSession`
6. Frames flow back to app through `ICameraDeviceCallbacks`

✓ References You Can Check in AOSP:

- `frameworks/av/services/camera/libcameraservice/ CameraService.cpp`
- `frameworks/av/camera/aidl/android/hardware/`
- `hardware/interfaces/camera/provider/2.4/` (or latest)
- `hardware/interfaces/camera/device/3.4/` (or latest)

Let me know if you'd like:

- A **step-by-step camera open and capture flow**
- Sample HIDL interface implementation
- A comparison of Camera HAL v1 vs v3
- Diagram annotated with actual function names in the flow

I'm happy to guide further depending on your goal (e.g., HAL development, debugging, testing).



* `ICameraService` * `ICameraDeviceUser`
explain

Great! Let's break down `ICameraService` and `ICameraDeviceUser`, two of the most important **AIDL Binder interfaces** in the Android camera architecture.

🔴 ICameraService: Manages all cameras

📍 Location:

`frameworks/av/camera/aidl/android/hardware/ICameraService.aidl`

✓ Purpose:

Acts as the **entry point** for camera access in the Android system.

It:

- Enumerates available camera devices
- Lets apps open a camera
- Handles camera status notifications
- Manages listener registration (e.g., for camera plug/unplug events)

📦 Think of it as:

The **camera system manager** that your app interacts with **before** a specific camera device is opened.

📌 Key Methods:

Method	Purpose
<code>getNumberOfCameras()</code>	Get how many cameras are available. (<i>deprecated</i>)
<code>getCameraCharacteristics()</code>	Return metadata for a camera (resolution, lens info, etc.)
<code>connectDevice()</code>	Connects to a specific camera → returns ICameraDeviceUser
<code>addListener()</code>	Register a listener (ICameraServiceListener) to get camera availability events

🔴 **ICameraDeviceUser: Controls one camera**

📍 Location:

`frameworks/av/camera/aidl/android/hardware/ICameraDeviceUser.aidl`

✓ Purpose:

Once a camera is opened (via **ICameraService**), you get an **ICameraDeviceUser** interface.

This interface:

- Sends capture requests to HAL
- Manages streams and buffers
- Receives image data

- Controls camera settings

💡 Think of it as:

The **per-camera connection handle** that the app uses to **send requests and receive results**.

📌 Key Methods:

Method	Purpose
<code>submitRequest()</code>	Send one or more capture requests to HAL
<code>createStream()</code>	Setup a new output stream (e.g., preview or JPEG)
<code>deleteStream()</code>	Remove a stream
<code>beginConfigure()/endConfigure()</code>	Define a group of streams together (batch setup)
<code>flush()</code>	Cancel all in-flight captures
<code>disconnect()</code>	Tear down the camera session

📈 Visual Flow: How They Relate

mermaid

```

sequenceDiagram
    App ->> ICameraService: openCamera("0")
    ICameraService -->> App: returns ICameraDeviceUser binder
    App ->> ICameraDeviceUser: createStream(), submitRequest(), etc.
    ICameraDeviceUser ->> CameraService: native camera operation
    CameraService ->> HAL: hardware control
    HAL -->> CameraService: image + metadata
    CameraService ->> ICameraDeviceCallbacks: results to app
  
```

🔒 Security Notes

- These interfaces cross **process boundaries**, so all data passed between them must be validated.
- **ICameraDeviceUser** is usually used by **CameraDeviceClient.cpp** inside the native camera service.

Summary

Interface	Role	Used By	Main Actions
ICameraService	Camera manager	Apps → Framework	Enumerate, open, listen
ICameraDeviceUser	Per-device control	Framework → HAL	Submit requests, manage streams

Would you like to see the real AIDL file or C++ implementation behind these?



- * `ICameraProvider` (lists available devices)
 - * `ICameraDevice` (represents one hardware camera)
 - * `ICameraDeviceSession` (active streaming/capture session)
- explain this in simple words with example

Absolutely! Let's explain these camera HAL components in **very simple terms** using a **real-world example** – like a **security guard system with walkie-talkies and a control room**.

Imagine this scenario:

You're running a big **building security system**:

- There are **multiple security cameras** on the roof, lobby, parking, etc.
- There's a **control room** where staff choose which camera to use and start watching live footage.
- The cameras talk through a **walkie-talkie system** to tell the control room what they're seeing.

Now, mapping that to Android's camera HAL:

Real-world thing	Android Component
Building control room	CameraService
Walkie-talkie system	Binder + HIDL interfaces
List of available cameras	ICameraProvider
One specific camera	ICameraDevice
A live viewing session	ICameraDeviceSession

1. **ICameraProvider** – List the cameras

What it is:

A HIDL interface that lists all available physical camera devices.

Think of it like:

The **security camera catalog** in the control room. It tells you:

- We have 3 cameras: Lobby (ID: 0), Roof (ID: 1), Parking (ID: 2)
- Which ones are plugged in, disconnected, or added dynamically

Simple example:

```
C++

ICameraProvider->getCameraIdList();
// returns: ["0", "1", "2"]
```

 **2. ICameraDevice -  Represents one camera****What it is:****The interface for a specific physical camera device.****Think of it like:**You pick **one camera** from the list (say, the **Roof Camera**) and grab its remote control.

It tells you:

- What the camera supports (resolution, flash, autofocus)
- Whether it can stream or record
- Opens a session when you're ready to use it

Simple example:

```
C++

ICameraProvider->getCameraDevice("1") → ICameraDevice
ICameraDevice->open() → ICameraDeviceSession
```

 **3. ICameraDeviceSession -  Controls live session****What it is:****The interface to actively control a camera stream**, send capture requests, and receive results.**Think of it like:**Once you **open the camera**, now you're in a **live call** with it.

You tell it:

- "Start preview"
- "Take a photo"
- "Send image frame data"

It responds:

- "Here's the frame"

- “Focus locked”
- “Capture complete”

Simple example:

c++

```
ICameraDeviceSession->configureStreams(...);
ICameraDeviceSession->processCaptureRequest(...);
```

Summary with Example Flow

text

App opens camera "1" (Roof Camera)

→ CameraService asks ICameraProvider: "What cameras are there?"
 → ICameraProvider replies: ["0", "1", "2"]
 → App chooses "1"
 → ICameraProvider gives back ICameraDevice for camera "1"
 → ICameraDevice opens → returns ICameraDeviceSession
 → App uses ICameraDeviceSession to:
 - Start preview
 - Take photo
 - Get image frames

Final Analogy Recap:

Android Term	Real World
ICameraProvider	Camera catalog
ICameraDevice	Individual camera and its remote
ICameraDeviceSession	Live camera stream controller

Let me know if you want the **actual function names and code** for each of these!



These three are different hals?

Great question!  Let's clarify:

No, these are not three different HALs.

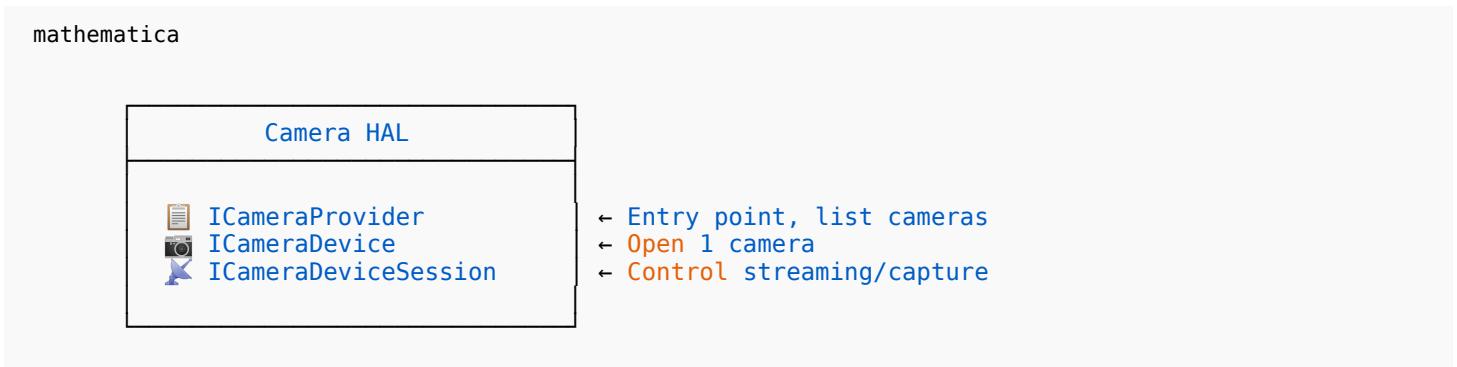
They are **three parts (or interfaces)** of the **same Camera HAL**, working together to implement the full camera pipeline.

In Simple Terms

You can think of the **Camera HAL** as a **team** with 3 members, each doing a specific job — but **all are part of the same HAL implementation**.

Interface	Role	Part of same HAL?	Defined in
ICameraProvider	Lists cameras, monitors status	✓ Yes	HIDL (AIDL after Android 12)
ICameraDevice	Controls a specific camera	✓ Yes	HIDL
ICameraDeviceSession	Runs capture sessions	✓ Yes	HIDL

Visualization: One HAL, Three Doors



Flow Between Them

1. **CameraService** talks to **ICameraProvider**
→ “What cameras are connected?”
2. **ICameraProvider** returns a list
→ e.g., `["0", "1", "2"]`
3. App picks camera “0”
→ `ICameraProvider::getCameraDevice("0")`
→ Returns **ICameraDevice**
4. **ICameraDevice::open()**
→ Returns **ICameraDeviceSession**
5. Use **ICameraDeviceSession**
→ To start preview, send capture requests, get frames

Are They Implemented Together?

Yes.

In most SoC vendor implementations (e.g., Qualcomm, MediaTek), the **same HAL library (`libcamera.so`)** implements all 3.

They are just **modular interfaces**:

- Defined using **HIDL** (or AIDL in newer Android versions)
- Loaded dynamically by **CameraService**

📍 Summary:

Question	Answer
Are these 3 different HALs?	<input checked="" type="checkbox"/> No
Are they part of the same HAL?	<input checked="" type="checkbox"/> Yes
Why are there 3 interfaces?	To divide responsibilities

Let me know if you want the source tree layout or a sample skeleton code for these HAL interfaces!



camera provider vs camera service

Great question! Let's break down **Camera Provider** vs **Camera Service** — they sound similar, but they serve **very different roles** in Android's camera architecture.

⌚ TL;DR Table

Feature	CameraProvider	CameraService
Where it lives	Part of the HAL (vendor side)	Part of Android framework (system side)
Runs in	Vendor process (e.g., provider@2.4-service)	System process (cameraservice)
Implements	ICameraProvider HIDL or AIDL interface	CameraService.cpp
Purpose	Talks directly to camera hardware	Talks to apps and manages camera access
Provides	Access to ICameraDevice , sessions, metadata	App-facing APIs via AIDL/Binder (ICameraService)
Loaded by	Dynamically loaded by CameraService	Booted with Android

📦 What is CameraProvider?

The **Camera Provider** is the entry point into the **Camera HAL**. It's implemented by the **hardware vendor**.

Role:

- Provides the list of available cameras
- Connects to specific hardware devices (e.g., sensors, ISPs)
- Creates **ICameraDevice** and **ICameraDeviceSession**
- Wraps all HAL logic and driver access

Code lives in:

`hardware/interfaces/camera/provider/`

Examples: `CameraProvider.cpp`, `ICameraProvider.hal`

What is CameraService?

The **Camera Service** is a **system service** in Android.

It's the **main brain** that connects:

- Apps \rightleftarrows HAL
- Java APIs \rightleftarrows native interfaces \rightleftarrows hardware

Role:

- Receives calls from apps (`CameraManager`, `CameraDevice`)
- Uses Binder interfaces like `ICameraService`, `ICameraDeviceUser`
- Loads the `CameraProvider`
- Enforces permission and security (e.g., one app at a time)
- Handles multi-client coordination

Code lives in:

`frameworks/av/services/camera/libcameraservice/`

Real Flow Example (Step by Step)

App wants to use the camera:

1. App calls:

`CameraManager.openCamera("0")`

2. CameraManager uses AIDL to call:

`ICameraService::connectDevice(...)`

3. CameraService does:

- Loads `CameraProvider` HAL
- Asks: `ICameraProvider::getCameraIdList()`
- Gets `ICameraDevice`
- Opens device, creates session

4. Session begins:

Camera frames flow from HAL → CameraService → App

🔌 Analogy

Item	CameraService	CameraProvider
Like a Manager	Controls who uses camera	Connects to camera hardware
Owned by	Android Framework (Google)	Hardware Vendor (Qualcomm, MTK, etc.)
Talks to	Apps & HAL	HAL → Camera driver

✓ Summary

- **CameraService** = **System-level boss** that manages all apps and camera access
- **CameraProvider** = **Vendor-level interface** that actually **talks to the camera hardware**
- **CameraService loads and uses CameraProvider** to fulfill requests from apps

Let me know if you want:

- A visual flow diagram
- Source file locations
- Or logs from `logcat` when this interaction happens!



camera server vs camera service

Great question! The terms **CameraServer** and **CameraService** sound similar, but they refer to **different things**, even though they're closely related.

Let's break it down simply:

✓ TL;DR Difference

Term	CameraService	CameraServer
What is it?	A C++ class that manages all camera logic	A system process that runs CameraService
Lives in	<code>CameraService.cpp</code> (native framework)	<code>/system/bin/cameraserver</code> (process on device)
Role	Provides camera APIs to apps and manages HAL	Hosts the CameraService in a separate sandbox
Used by	Framework & apps via AIDL/Binder	Android system (runs in background)
Security Role	Core logic for access control and camera ops	Isolated service process with lower privileges

📘 CameraService (C++ class)

- It's the **main controller** of the Android camera system.
- Handles:
 - Listing cameras
 - Opening/closing them
 - Coordinating apps' access
 - Sending capture requests to HAL

📁 Code location:

`frameworks/av/services/camera/libcameraservice/CameraService.cpp`

It implements:

- **ICameraService** (AIDL interface)
- Owns and talks to:
 - **CameraProviderManager**
 - **CameraDeviceClient**
 - **Camera3Device**

⚙️ CameraServer (process)

- It is the **process** that runs the **CameraService** class.
- Starts as a **separate Linux process** for security reasons:
 - Can be sandboxed
 - Runs with reduced privileges (e.g., no access to system files)

 Executable:

`/system/bin/cameraserver`

 Source code:

`frameworks/av/camera/cameraserver/`

 Introduced in Android 7.0+ as part of **media server split** for **security hardening**.

Analogy

Real-world thing	What it represents
<code>CameraService</code> class	The camera manager software
<code>cameraservice</code> process	The room/computer running that software

Just like a **banking app (code)** runs inside your **phone app process**,
`CameraService` (code) runs inside `cameraservice` (process).

How they work together

1. System boots
2. Android starts `cameraservice` process
3. Inside it, `CameraService` class is instantiated
4. Apps talk to `CameraService` via Binder
5. `CameraService` talks to the HAL via `CameraProvider`

How to observe this

Run this on a device:

```
bash
ps -A | grep cam
```

You'll see something like:

```
bash
media    1234  ... /system/bin/cameraserver
```

And check logs:

```
bash
logcat | grep CameraService
```

✓ Summary

Term	CameraService	CameraServer
What is it?	C++ service logic	Linux process
Purpose	Manages cameras	Hosts the service securely
Code	CameraService.cpp	main_cameraserver.cpp
Privileges	High (internal control)	Low (sandboxed)

Let me know if you'd like a flowchart showing how **CameraManager** → **CameraService** → **Camera HAL** works!



how to check the cameraservice usage

You can check the usage and behavior of **CameraService** on a running Android device using various tools and logs. Here's a simple and complete guide:

✓ 1. Check if CameraService is running

Open a shell to the device:

```
bash
adb shell
ps -A | grep cam
```

You should see:

```
bash
media    1234  ... /system/bin/cameraserver
```

This confirms the **cameraserver process** is running, which hosts **CameraService**.

✓ 2. Check logs from CameraService

To monitor logs when apps use the camera:

```
bash  
adb logcat | grep CameraService
```

You'll see logs like:

```
mathematica  
  
I CameraService: CameraService started (pid=1234)  
I CameraService: connectDevice: E  
I CameraService: connectDevice: X  
I CameraService: Camera device opened
```

These show when:

- CameraService starts
- A camera is opened or closed
- Clients (apps) connect or disconnect

✓ 3. Check who is using the camera

Run this to find camera usage via logcat:

```
bash  
adb logcat | grep -i "camera"
```

Or more specifically:

```
bash  
adb shell dumpsys media.camera
```

You'll see output like:

```
yaml  
  
CameraService:  
    Number of camera devices: 2  
    Active Camera Clients:
```

```
- Camera ID: 0
  Client: com.example.mycameraapp (PID 2563, UID 10123)
  API Level: 2
  Device: active
```

This tells you:

- Which apps are using which camera
- What API level they're using
- If the device is in use or idle

✓ 4. Monitor binder usage (who's calling CameraService)

```
bash
adb shell dumpsys activity service cameraservice
```

Or check all binder services:

```
bash
adb shell service list | grep -i camera
```

✓ 5. Get more detailed internal status

```
bash
adb shell dumpsys media.camera | less
```

Look for sections like:

```
yaml
Client:
  Package name: com.android.camera
  API Level: 2
  PID: 2345
  UID: 10123
  Client package is trusted: true
  Device active: YES
```

✓ 6. Bonus: Trace camera open/close events

You can trace app camera usage live with:

```
bash
adb shell atrace -c; adb shell atrace -b 4096 -t 10 camera
```

This gives a 10-second trace showing camera usage from framework down to HAL level.

📌 Summary

Task	Command
Check if cameraserver is alive	`ps -A
Log camera activity	`adb logcat
See who is using camera	adb shell dumpsys media.camera
List camera services	`adb shell service list
Trace usage live	adb shell atrace -t 10 camera

Let me know if you want to script or automate camera usage monitoring!



Legacy HAL components

This section describes the architecture of the legacy HAL components and how to implement the HAL. Camera HAL implementations on Android 8.0 and higher must use the HIDL API instead, described above.

Architecture (legacy)

The following figure and list describe the legacy camera HAL components.

Great! Let's walk through the **Legacy Camera HAL** architecture in Android — which was used **before Android 8.0 (Oreo)** — and compare it with the modern HIDL-based approach.

📦 Legacy Camera HAL Overview (pre-Android 8.0)

Before Android 8.0, camera HALs used a **non-binderized**, simple **C-style interface** with function pointers and structs.

🔧 Key Characteristics

Feature	Legacy HAL (pre-8.0)
Interface style	C structs and function pointers

Feature	Legacy HAL (pre-8.0)
HAL entry point	<code>HAL_MODULE_INFO_SYM</code>
Core header	<code>hardware/camera.h</code>
Not binderized	<input checked="" type="checkbox"/> Yes — Direct function calls
Used in	<code>Camera HAL v1, Camera HAL v2</code>
Replaced by	HIDL-based HAL in Android 8.0+

📁 Typical Legacy HAL Structure

1. `camera_module_t`

- The main entry point for the camera HAL.
- Contains metadata about the HAL (e.g., number of cameras, name, version).
- Must export a global symbol: `HAL_MODULE_INFO_SYM`

```
c

extern "C" {
    struct hw_module_t HAL_MODULE_INFO_SYM = {
        .id = CAMERA_HARDWARE_MODULE_ID,
        .name = "MyCameraHAL",
        .methods = &camera_module_methods,
        ...
    };
}
```

2. `camera_module_methods_t`

- Contains function pointers like `open()`:

```
c

static hw_module_methods_t camera_module_methods = {
    .open = my_camera_device_open,
};
```

3. `camera_device_t`

- Represents a single camera.
- Implements function pointers for starting preview, taking pictures, etc.

c

```
camera_device_t* device;
device->ops->start_preview();
device->ops->take_picture();
```

4. HAL Implementation Functions

Examples of legacy function hooks you had to implement:

```
c

int camera_device_open(...) {
    // Allocate and initialize camera device
}

int camera_start_preview(camera_device_t* dev) {
    // Start sending preview frames
}

int camera_take_picture(camera_device_t* dev) {
    // Capture JPEG frame
}
```

These functions directly interacted with the hardware (sensor, ISP, etc.).

Legacy Camera Flow Summary

scss

```
[ App ]
  ↓
[ Camera Framework ]
  ↓ (direct function calls)
[ camera_module_t ]
  ↓
[ camera_device_t ]
  ↓
[ Driver (/dev/videoX) ]
```

No Binder, no HIDL — just function calls between framework and vendor code.

Why Legacy HAL Was Deprecated

Problem	Impact
No security boundary	Framework trusted HAL blindly
No IPC isolation	All HALs ran in same process space
No versioning	Difficult to upgrade without breaking

Problem	Impact
Not testable or modular	No mocking or separation of concerns

✓ Modern Approach (Post Android 8.0+)

Feature	Legacy HAL	HIDL-based HAL
Interface style	Function pointers	Binderized HIDL (now AIDL)
Process isolation	✗ No	✓ Yes
Secure boundary	✗ No	✓ Yes (HALs are sandboxed)
Dynamic loading	Manual linking	Via ICameraProvider
Used by	HALv1/HALv2	HALv3+

✓ Summary

Term	Description
Legacy Camera HAL	Non-binderized, C-style, used pre-Android 8
camera_module_t	Main entry struct for the HAL
camera_device_t	Represents a single opened camera
Function pointers	Implement preview, capture, focus, etc.
Replaced by	HIDL-based Camera HAL in Android 8.0 and later

Would you like:

- A **sample implementation** of a legacy HAL?
- Comparison of **camera_device_ops_t** vs **ICameraDeviceSession**?
- Or how to wrap a legacy HAL inside a HIDL provider (common for backward compatibility)?



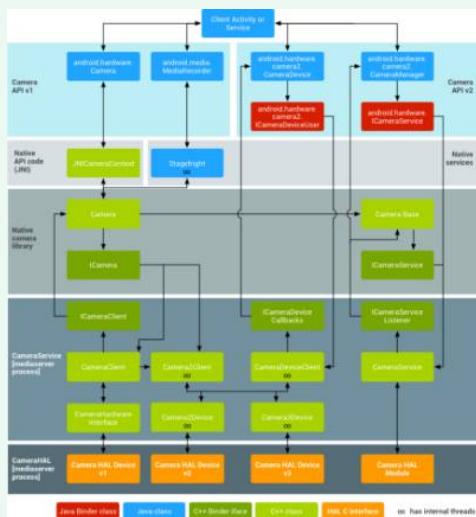


Figure 2. Legacy camera architecture

app framework

At the app framework level is the app's code, which uses the `android.hardware.Camera` API to interact with the camera hardware. Internally, this code calls a corresponding JNI glue class to access the native code that interacts with the camera.

JNI

The JNI code associated with `android.hardware.Camera` is located in `frameworks/base/core/jni/android_hardware_Camera.cpp`. This code calls the lower-level native code to obtain access to the physical camera and returns data that is used to create the `android.hardware.Camera` object at the framework level.

native framework

The native framework defined in `frameworks/av/camera/Camera.cpp` provides a native equivalent to the `android.hardware.Camera` class. This class calls the IPC binder proxies to obtain access to the camera service.

binder IPC proxies

The IPC binder proxies facilitate communication over process boundaries. There are three camera binder classes that are located in the `frameworks/av/camera` directory that calls into camera service. `ICameraService` is the interface to the camera service, `ICamera` is the interface to a specific opened camera device, and `ICameraClient` is the device's interface back to the app framework.

camera service

The camera service, located in `frameworks/av/services/camera/libcameraservice/CameraService.cpp`, is the actual code that interacts with the HAL.

HAL

The hardware abstraction layer defines the standard interface that the camera service calls into and that you must implement to have your camera hardware function correctly.

kernel driver

The camera's driver interacts with the actual camera hardware and your implementation of the HAL. The camera and driver must support YV12 and NV21 image formats to provide support for previewing the camera image on the display and video recording.

Implement the HAL (legacy)

The HAL sits between the camera driver and the higher-level Android framework and defines an interface that you must implement so apps can correctly operate the camera hardware. The HAL interface is defined in the hardware/libhardware/include/hardware/camera.h and hardware/libhardware/include/hardware/camera_common.h header files.

camera_common.h defines camera_module, a standard structure to obtain general information about the camera, such as the camera ID and properties common to all cameras (that is, whether it is a front- or back-facing camera).

camera.h contains code that corresponds to android.hardware.Camera. This header file declares a camera_device struct that in turn contains a camera_device_ops struct with pointers to functions that implement the HAL interface. For documentation on the camera parameters developers can set, refer to frameworks/av/include/camera/CameraParameters.h. These parameters are set with the function pointed to by int (*set_parameters)(struct camera_device *, const char *parms) in the HAL.

For an example of a HAL implementation, refer to the implementation for the Galaxy Nexus HAL in hardware/ti/omap4xxx/camera.

Configure the shared library

Set up the Android build system to correctly package the HAL implementation into a shared library and copy it to the appropriate location by creating an Android.mk file:

Create a device/<company_name>/<device_name>/camera directory to contain your library's source files.

Create an Android.mk file to build the shared library. Ensure that the makefile contains the following lines:

```
LOCAL_MODULE := camera.<device_name>
LOCAL_MODULE_RELATIVE_PATH := hw
```

Your library must be named camera.<device_name> (.so is appended automatically), so Android can correctly load the library. For an example, see the makefile for the Galaxy Nexus camera located in hardware/ti/omap4xxx/Android.mk.

Specify your device has camera features by copying the necessary feature XML files in the frameworks/native/data/etc directory with your device's makefile. For example, to specify your device has a camera flash and can autofocus, add the following lines in your device's <device>/<company_name>/<device_name>/device.mk makefile:

```
PRODUCT_COPY_FILES := \ ...
```

```
PRODUCT_COPY_FILES += \
frameworks/native/data/etc/android.hardware.camera.flash-
autofocus.xml:system/etc/permissions/android.hardware.camera.flash-autofocus.xml \
For an example of a device makefile, see device/samsung/tuna/device.mk.
```

Declare your camera's media codec, format, and resolution capabilities in device/<company_name>/<device_name>/media_profiles.xml and device/<company_name>/<device_name>/media_codecs.xml XML files. For details, see Exposing codecs to the framework.

Add the following lines in your device's device/<company_name>/<device_name>/device.mk makefile to copy the media_profiles.xml and media_codecs.xml files to the appropriate location:

```
# media config xml file
PRODUCT_COPY_FILES += \
    <device>/<company>/<device>/media_profiles.xml:system/etc/media_profiles.xml
```

```
# media codec config xml file
PRODUCT_COPY_FILES += \
    <device>/<company>/<device>/media_codecs.xml:system/etc/media_codecs.xml
```

To include the Camera app in your device's system image, specify it in the PRODUCT_PACKAGES variable in your device's device/<company>/<device>/device.mk makefile:

```
PRODUCT_PACKAGES := \
    Gallery2 \
    ...
```

Great! You're diving into the **legacy camera HAL architecture**, and the diagram you've provided helps visualize the full path from an Android camera app to the physical camera hardware. Let's break it down **in simple terms** and explain how the pieces connect.

Big Picture: How Android Camera Works (Legacy - Pre Android 8.0)

Imagine you're using a camera app like Instagram or Snapchat. Here's what happens behind the scenes to take a picture:

◆ 1. App Framework (Top Blue Layer)

This is the **Java** layer where Android apps live.

- **android.hardware.Camera**: Older API (Camera API v1)
- **android.media.MediaRecorder**: For recording videos
- **android.hardware.camera2.CameraDevice**: Newer API (Camera2 API)
- **android.hardware.camera2.CameraManager**: Lists available cameras

These APIs let apps control the camera (e.g., open, focus, capture).

2. JNI Layer (Green - Native Interface)

JNI (Java Native Interface) connects Java code to **native C++** code.

- **JNI_Context & Stagefright** pass the app's commands down to native code (e.g., open camera, start preview).

3. Native Services (Native camera library)

This is **C++ code** that talks to the camera service.

- **Camera, ICamera, ICameraService**: Native interfaces matching the Java ones
- **ICameraClient**: Used by the framework to get camera updates (e.g., preview frame is ready)

4. CameraService (in mediaserver process)

Think of this as the "**brain**" of Android's camera system. It manages:

- Opening and closing cameras
- Routing requests from apps
- Managing sessions and callbacks

It uses these client classes:

- **CameraClient** (for API v1)
- **Camera2Client** (for Camera2)
- **CameraDeviceClient** (API v2 apps)

Each client talks to its respective **CameraXDevice** (HAL v1, v2, v3).

5. Camera HAL (in mediaserver)

This is where **hardware control** happens.

- The HAL (Hardware Abstraction Layer) is a set of C function pointers that Android calls to talk to your camera hardware.
- Different versions:

- Camera HAL v1: For `CameraClient`
- Camera HAL v2: For `Camera2Client`
- Camera HAL v3: For `CameraDeviceClient`

Defined in:

- `camera.h`, `camera_common.h`

Example: The HAL implementation says **how to actually open the camera, start the preview, take a photo, etc.**

6. Driver & Hardware

At the lowest level:

- The **driver** is part of the Linux kernel.
 - It communicates with the actual **camera sensor hardware**.
 - The driver supports formats like YV12 and NV21 (for preview/video).
-

Example: Taking a Picture

Let's say you call `camera.takePicture()` in your app. Here's what happens:

1. Java app calls `android.hardware.Camera.takePicture()`
 2. JNI calls native `Camera` class
 3. Native code uses `ICamera` Binder interface
 4. Binder IPC calls into `CameraService`
 5. `CameraService` uses `CameraClient` to talk to `CameraHardwareInterface`
 6. `CameraHardwareInterface` calls `camera_device->ops->take_picture()`
 7. HAL implementation interacts with kernel driver
 8. Driver captures a JPEG image and returns it up the stack
 9. You get a picture callback in your app!
-

Summary Table

Layer	Role
App Framework (Java)	Camera API used by app developers

Layer	Role
JNI Layer	Bridge from Java to native (C++) code
Native Services	C++ interfaces to camera functionality
CameraService	Main logic hub that routes and manages camera usage
Camera HAL	Vendor code to control real camera hardware
Driver & Hardware	Physical camera and Linux kernel driver

Let me know if you'd like:

- A **code-level breakdown** of `camera_device_ops`
- A comparison between **HAL v1 and v3**
- Or how to **migrate from legacy HAL to HIDL HAL**

I'm happy to walk you through any part step-by-step!



camera3 HAL vs Legacy HAL

Great question! Let's break this down **clearly and simply** using your diagram and Android's HAL evolution.

✓ 1. What is Legacy Camera HAL (camera1 HAL)?

The **Legacy HAL** is the **old camera HAL** used in **Android 4.4 and below**, and sometimes in backward compatibility (called **LEGACY mode** in Camera2).

* Main Features:

- Simple interface
- Used with the **old Camera API** (`android.hardware.Camera`)
- Synchronous and blocking (e.g., take picture, wait)
- No real-time metadata
- Single stream (preview OR capture, not both well)

✓ 2. What is Camera3 HAL (camera HAL v3)?

Camera3 HAL is used in **Android 5.0 (Lollipop)** and above, and powers the **Camera2 API** (`android.hardware.camera2`).

* Main Features:

- Asynchronous pipeline
- Real-time metadata support (like autofocus state, exposure info)
- Supports multiple streams (preview + video + capture simultaneously)
- Supports manual controls (ISO, exposure, focus, etc.)
- Designed for **high-performance, multi-buffer, low-latency** use cases

📊 Comparison Table

Feature	Legacy HAL (Camera HAL v1)	Camera3 HAL (Camera HAL v3)
Android Version	≤ Android 4.4	Android 5.0+
Used By	Camera API	Camera2 API
Architecture	Simple, blocking	Asynchronous pipeline
Metadata	Minimal	Rich metadata per frame
Multi-stream	Poor	Excellent
Manual control	Limited	Full manual control
Performance	Lower	High performance
Replaced By	Camera HAL v3	Still current

📌 Real-World Example

Let's say you're building a camera app:

- 📱 On **old phones (legacy HAL)**: You can only do preview OR capture. Autofocus or exposure data is hard to get.
- 📱 On **modern phones (Camera HAL v3)**: You can show live preview, record video, and capture photos **at the same time**, while adjusting focus, ISO, white balance in real time.

🔄 Transition

- When Android introduced **Camera2 API**, vendors were required to implement **camera3 HAL**.
- If they didn't, the system used a **Legacy wrapper** to convert old HAL behavior to fit Camera2 needs (but with reduced features).

✓ Summary

- **Legacy HAL:** Simple, used with old APIs, lacks modern features.
- **Camera3 HAL:** Complex, powerful, supports modern apps and features.

Let me know if you want to see actual code differences or how they appear in logs.



so when camera binderized hal came

Great follow-up! Let's walk through the evolution and pinpoint **when the camera HAL became binderized**.

📅 Timeline of Camera HAL Evolution

Android Version	HAL Type	Key Characteristics
≤ Android 4.4	Legacy Camera HAL v1	Simple C-style API, direct in-process calls, no binder
Android 5.0	Camera HAL v3 (camera3)	For Camera2 API , still in-process, not binderized
Android 8.0	Binderized via HIDL	Camera HAL is now binderized using HIDL
Android 11+	AIDL-based HAL (optional)	Gradual shift to AIDL from HIDL, but not yet for camera HAL

📌 So when did Camera HAL become binderized?

Camera HAL became binderized in **Android 8.0 (Oreo)** using **HIDL (HAL Interface Definition Language)**.

🔍 What does "binderized" mean here?

Before Android 8.0:

- HALs were shared libraries (.so files) loaded **directly into system_server or cameraserver**.
- It was **in-process**, meaning crashes in HAL could crash the entire system.

After Android 8.0:

- HALs (like camera) became **Binderized Services**, running in **separate processes** (e.g., `android.hardware.camera.provider@2.4-service`).

- Communicated via **HIDL over Binder IPC**.
- Safer: If the HAL crashes, only that service dies, not the system server.

✓ Components introduced with binderization:

1. android.hardware.camera.provider@2.X (HIDL)

- Entry point for the framework to enumerate and open camera devices.

2. android.hardware.camera.device@3.X (HIDL)

- Defines camera3 HAL device operations (stream setup, capture requests, etc.)

3. camerасerver

- Moved out of `system_server` in Android 7.0
- Talks to the HIDL-based HAL via binder.

🔧 How to check if camera HAL is binderized on a device:

```
bash
```

```
adb shell lshal | grep camera
```

You should see something like:

```
ruby
```

```
android.hardware.camera.provider@2.4: :ICameraProvider/legacy/0
```

That means the HAL is running as a **HIDL binderized service**.

✓ Summary

Question	Answer
When did camera HAL become binderized?	Android 8.0 (Oreo)
Which technology?	HIDL over Binder IPC
Why?	To improve stability, security, and modularity
What's next?	Android is migrating some HALs to AIDL , but camera HAL is still mostly HIDL

Let me know if you'd like to see a visual layout of how the camera HAL interacts across processes before and after binderization.



camera server vs system server

Great question — understanding the difference between **cameraservice** and **system_server** is key to grasping Android's architecture evolution, especially post-Android 7.0. Let's break it down clearly:

✓ 1. What is system_server?

📌 Purpose:

system_server is the **central Java system process** in Android that hosts most core system services (e.g., **ActivityManagerService**, **PackageManagerService**, etc.).

🧠 Key facts:

- Runs **Java services**.
- Loads **framework code** (e.g., `CameraManager.java`).
- Prior to Android 7.0, it directly loaded **native HALs**, including camera.

✓ 2. What is cameraservice?

📌 Purpose:

cameraservice is a **native (C++) service process** that manages all interactions with the **Camera HAL**.

🧠 Key facts:

- Introduced in **Android 7.0 (Nougat)**.
- Separated from **system_server** for **security and stability**.
- Talks to the Camera HAL via **Binder** (if HAL is binderized, i.e. HIDL/AIDL).
- Runs **CameraService** (native C++).
- Avoids crashing the entire system if camera HAL fails.

🧱 Summary: system_server VS cameraservice

Feature	system_server	cameraservice
Language	Java	Native C++
Purpose	Runs system-level Java services	Hosts native CameraService and HAL access
Camera Responsibility	Used to control camera HAL pre-Android 7.0	Fully manages HAL from Android 7.0+
Crash Risk (pre-A7)	HAL crash → brings down system_server	Isolated crash in cameraservice only
Security Context	u:r:system_server:s0	u:r:cameraservice:s0

🎯 Why was cameraservice introduced?

Before Android 7.0:

- Camera HAL ran **in-process** inside **system_server**.
- Bugs or malicious drivers could **crash or exploit the entire OS**.

After Android 7.0:

- Moved camera access to **cameraservice** (separate sandboxed process).
- Enhanced **privilege separation, SELinux confinement, and stability**.

🔍 How do they interact?

1. App uses **CameraManager** (Java API).
2. **CameraManager** talks to **CameraService** via Binder (in **cameraservice**).
3. **CameraService** talks to the **Camera HAL** (binderized or legacy).
4. Results are returned back through the same path.

If you'd like, I can draw a process diagram showing this flow visually — just ask!



Camera HAL

bookmark_border

Android's camera hardware abstraction layer (HAL) connects the higher level camera framework APIs in android.hardware.camera2 to your underlying camera driver and hardware. Starting with Android 13, camera HAL interface development uses AIDL. Android 8.0 introduced Treble, switching the Camera HAL API to a stable interface defined by the HAL interface description language (HIDL). If you've previously developed a camera HAL module and driver for Android

7.0 and lower, be aware of significant changes in the camera pipeline.

AIDL camera HAL

For devices running Android 13 or higher, the camera framework includes support for AIDL camera HALs. The camera framework also supports HIDL camera HALs, however camera features added in Android 13 or higher are available only through the AIDL camera HAL interfaces. To implement such features on devices upgrading to Android 13 or higher, device manufacturers must migrate their HAL process from using HIDL camera interfaces to AIDL camera interfaces.

To learn about the advantages of AIDL, see [AIDL for HALs](#).

Implement AIDL camera HAL

For a reference implementation of an AIDL camera HAL, see [hardware/google/camera/common/hal/aidl_service/](#).

The AIDL camera HAL specifications are in the following locations:

Camera provider: [hardware/interfaces/camera/provider/aidl/](#)

Camera device: [hardware/interfaces/camera/device/aidl/](#)

Camera metadata: [hardware/interfaces/camera/metadata/aidl/](#)

Common data types: [hardware/interfaces/camera/common/aidl/](#)

For devices migrating to AIDL, device manufacturers might need to modify the Android SELinux policy (sepolicy) and RC files depending on the code structure.

Validate AIDL camera HAL

To test your AIDL camera HAL implementation, ensure that the device passes all CTS and VTS tests. Android 13 introduces the AIDL VTS test, `VtsAidlHalCameraProvider_TargetTest.cpp`.

Camera HAL3 features

The aim of the Android Camera API redesign is to substantially increase the ability of apps to control the camera subsystem on Android devices while reorganizing the API to make it more efficient and maintainable. The additional control makes it easier to build high-quality camera apps on Android devices that can operate reliably across multiple products while still using device-specific algorithms whenever possible to maximize quality and performance.

Version 3 of the camera subsystem structures the operation modes into a single unified view, which can be used to implement any of the previous modes and several others, such as burst mode. This results in better user control for focus and exposure and more post-processing, such as noise reduction, contrast and sharpening. Further, this simplified view makes it easier for application developers to use the camera's various functions.

The API models the camera subsystem as a pipeline that converts incoming requests for frame captures into frames, on a 1:1 basis. The requests encapsulate all configuration information about the capture and processing of a frame. This includes resolution and pixel format; manual sensor, lens and flash control; 3A operating modes; RAW->YUV processing control; statistics

generation; and so on.

In simple terms, the application framework requests a frame from the camera subsystem, and the camera subsystem returns results to an output stream. In addition, metadata that contains information such as color spaces and lens shading is generated for each set of results. You can think of camera version 3 as a pipeline to camera version 1's one-way stream. It converts each capture request into one image captured by the sensor, which is processed into:

A result object with metadata about the capture.

One to N buffers of image data, each into its own destination surface.

The set of possible output surfaces is preconfigured:

Each surface is a destination for a stream of image buffers of a fixed resolution.

Only a small number of surfaces can be configured as outputs at once (~3).

A request contains all desired capture settings and the list of output surfaces to push image buffers into for this request (out of the total configured set). A request can be one-shot (with `capture()`), or it may be repeated indefinitely (with `setRepeatingRequest()`). Captures have priority over repeating requests.

You've shared a detailed explanation of the **Camera HAL evolution**, especially the move from **legacy HALs** to **Camera HAL3**, and eventually to **AIDL-based Camera HALs** starting in **Android 13**. Let's break this down into **simpler terms**, compare the technologies, and summarize key takeaways.

What is Camera HAL? (Simple Definition)

The **Camera HAL (Hardware Abstraction Layer)** is a software layer that connects Android's high-level camera APIs (used by apps) to the **low-level camera drivers and hardware**.

- It acts like a **translator** between your camera app and the physical camera.
- Developers implement this layer so Android can control different camera hardware in a consistent way.

Evolution of Camera HAL Versions

Version	Android Version	Interface Type	Description
Legacy HAL v1	≤ Android 4.4	C interface	Simple, limited control. Used by Camera API (v1)
Camera HAL v3	Android 5.0+	C interface	Advanced features (manual focus, RAW, burst). Supports Camera2 API
HIDL Camera HAL	Android 8.0–12	HIDL (Binderized)	Modularized with Project Treble. Improved security and crash isolation.

Version	Android Version	Interface Type	Description
AIDL Camera HAL	Android 13+	AIDL	New features only here. Replaces HIDL gradually. Used by modern devices.

Camera HAL3 vs Legacy Camera HAL (v1)

Feature	Legacy HAL v1	Camera HAL v3 (HAL3)
API Support	<code>android.hardware.Camera</code>	<code>android.hardware.camera2</code>
Flexibility	Limited manual control	Full manual control (sensor, lens)
Output model	One-way stream	Capture pipeline with metadata
Capture types	Preview, snapshot	Preview, snapshot, burst , RAW
Metadata	Minimal or none	Rich metadata for each capture
Post-processing	Device-dependent	Explicit control (noise, sharpening)
Developer usage	Easy but limited	More powerful but complex

HIDL vs AIDL for Camera HAL

HIDL (Android 8-12)

- Defined HAL interfaces with `.hal` files
- Communicates via **Binder IPC**
- Good for modularity (Treble), but limited evolution

AIDL (Android 13+)

- HALs defined in `.aidl` files
- Easier to evolve (add fields/methods)
- New camera features (post Android 13) require AIDL**
- Preferred for new devices and features

Camera HAL3: The Capture Pipeline

Camera HAL3 introduced a **pipeline model**, making camera operation more robust and flexible:

How it works (Simple View):

1. App sends a **Capture Request** → Includes settings like resolution, exposure, focus, etc.

2. Camera HAL processes it → Captures an image

3. HAL returns:

- **Image Buffers** (to specified output surfaces like preview, JPEG, etc.)
- **Metadata** (focus distance, brightness, white balance, etc.)

Request Types:

- **One-shot:** `capture(request)`
- **Repeating:** `setRepeatingRequest(request)`

AIDL HAL Migration Tips

For device makers migrating to **AIDL**:

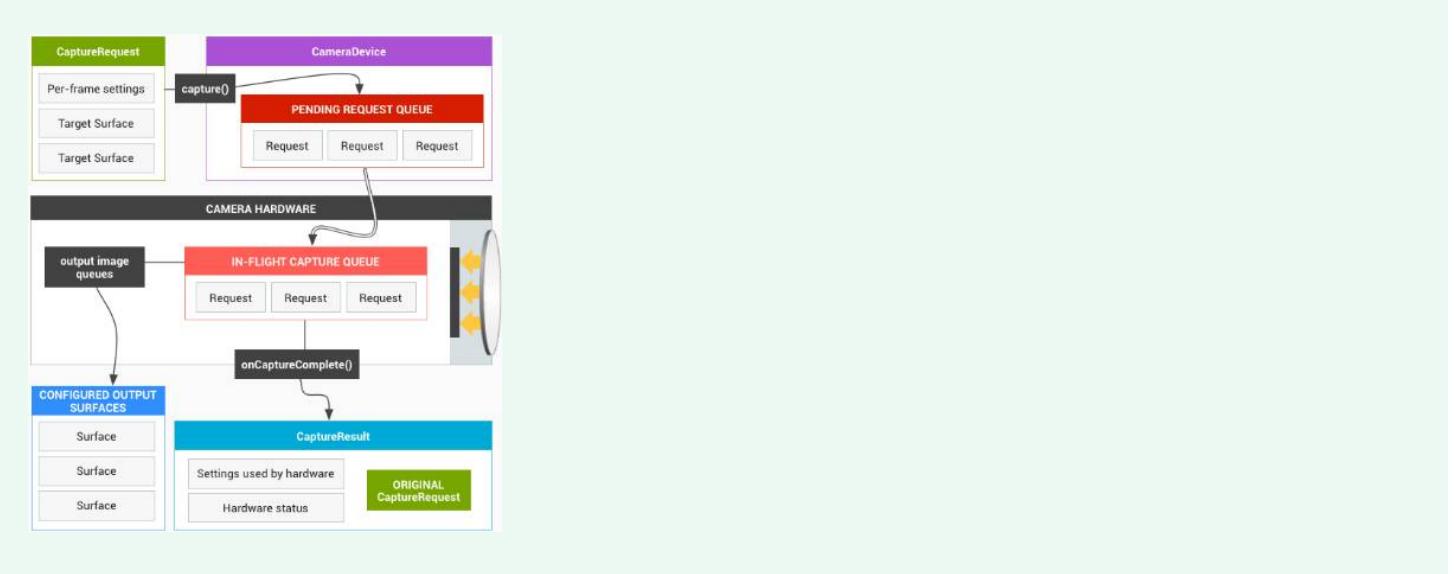
- Update your HAL implementation using:
 - `hardware/interfaces/camera/provider/aidl/`
 - `hardware/interfaces/camera/device/aidl/`
- Adjust **SEPolicy and RC files**
- Pass **CTS/VTS tests**, especially:
 - `VtsAidlHalCameraProvider_TargetTest.cpp`

Summary: What you need to know

Topic	Summary
Why HAL3 exists	To give apps more control and higher-quality image capture
Why AIDL now?	Easier to maintain and evolve; mandatory for Android 13+ features
HAL3 vs Legacy	HAL3 is a flexible, full-featured camera pipeline
AIDL vs HIDL	AIDL is the newer, cleaner IPC for HALs (HIDL is older)
Developer impact	Apps use Camera2 API ; HAL devs must follow AIDL for Android 13+

Would you like a **diagram** showing this evolution (Legacy → HAL3 → HIDL → AIDL) or a **real code example** of an AIDL-based Camera HAL interface?





The diagram you shared is a simplified model of the **Android Camera2 API** capture pipeline. Here's a breakdown of each part in the image and how they work together:

◆ 1. CaptureRequest

- This is the **app's request** to capture a frame.
- It includes:
 - **Per-frame settings** (e.g., exposure, focus)
 - **Target output Surfaces** (e.g., preview, JPEG, YUV stream)

◆ 2. CameraDevice & Pending Request Queue

- When the app calls `cameraDevice.capture(request)`, the request enters the **Pending Request Queue**.
- This is a **software-side queue** managed before the hardware actually begins processing.

🔧 3. Camera Hardware Layer

- The pending requests are moved into the **In-flight Capture Queue**, handled by the camera HAL and hardware.
- Here, each request is **actively processed** (sensor readout, ISP, etc.).

🟠 4. In-Flight Capture Queue

- This contains capture requests **currently being handled by the hardware**.
- It can hold multiple requests at once (like a pipeline).

5. Configured Output Surfaces

- These are **app-configured output destinations**, like:
 - `SurfaceView`, `ImageReader`, `MediaRecorder`
- The image data from hardware is sent here once the capture is processed.

6. CaptureResult

- After hardware finishes processing:
 - Metadata and settings used by the hardware are returned to the app.
 - Sent via a callback like `onCaptureCompleted()`.
 - The app can read these results for auto-focus, exposure, sensor info, etc.

Overall Flow Summary:

```
scss

App (CameraRequest)
  ↓
CameraDevice.enqueue(request)
  ↓
Hardware (process)
  ↓
Output images → Configured Surfaces
  ↓
Metadata → CaptureResult callback
```

This model is specific to **Camera HAL3**, introduced with the **Camera2 API** (Android 5.0+). It supports features like per-frame control, burst capture, RAW output, etc.

Let me know if you want a comparison between Camera HAL1 (legacy) and HAL3 or how this fits with binderized HAL architecture.



The number of queues used here ?

In the simplified Camera2 model you provided, there are **two main queues** involved in the capture pipeline:

1. Pending Request Queue

- **Location:** In the `CameraDevice` software layer (Framework side).
- **Purpose:**
 - Stores requests submitted by the app (`CaptureRequest`).
 - Queued before being sent to the HAL and hardware.
- **Controlled by:** Camera framework (`CameraDeviceImpl`, `Camera3Device`, etc.)
- **Example:**

```
java
cameraCaptureSession.capture(request, callback, handler);
```

This adds the request to the **pending queue**.

2. In-Flight Capture Queue

- **Location:** Inside the **Camera HAL3 implementation** and hardware pipeline.
- **Purpose:**
 - Holds requests that are currently being processed by the sensor and ISP.
 - Each request remains in this queue until processing is complete and metadata is available.
- **Controlled by:** Camera HAL3 + Driver
- **Includes:**
 - Frame readout
 - Image processing
 - Metadata generation

(Optional View) – Think of them as:

Queue Name	Acts As	Managed By
Pending Request Queue	"Waiting room" for frames	Camera framework
In-Flight Capture Queue	"Processing room"	HAL + hardware

Let me know if you want to dive into the actual AOSP classes where these queues are implemented (`Camera3Device`, `RequestThread`, etc.).



where buffers allocation happen?

Great question! In the Android Camera2 architecture (especially with **Camera HAL3**), **buffer allocation** is a key step and happens in a few specific places depending on the type of buffer and its owner. Here's a clear breakdown:

📍 Where buffer allocation happens:

◆ 1. Application / Framework allocates output buffers

- The app (or system service like `CameraService`) allocates **output buffers** via the `ImageReader`, `Surface`, or `SurfaceTexture` it provides.
- These are typically **gralloc** (`GraphicBuffer`) memory buffers.

Example:

- App creates an `ImageReader`:

```
java
ImageReader.newInstance(width, height, ImageFormat.YUV_420_888, maxImages);
```

► This allocates a **buffer queue** using the `GraphicBufferAllocator`.

- App creates a `SurfaceTexture`:
 - Internally creates a **buffer queue** managed by `BufferQueue`.

◆ 2. HAL receives and uses framework-allocated buffers

- The **Camera HAL3** doesn't allocate buffers itself in general.
- Instead, it receives buffer handles (e.g., `buffer_handle_t`) **via the request sent by framework**.
- It uses these buffers for image writing (e.g., JPEG, RAW, YUV).

◆ 3. Internal scratch or metadata buffers (HAL side)

- HAL may allocate its own **temporary buffers** for:
 - Reprocessing
 - Intermediate conversion (e.g., Bayer to YUV)
 - Metadata
- These are usually done via:

- `mmap`, `ion`, or `dmabuf` (depending on platform)
- `android::GraphicBufferAllocator`

🧠 Example Flow

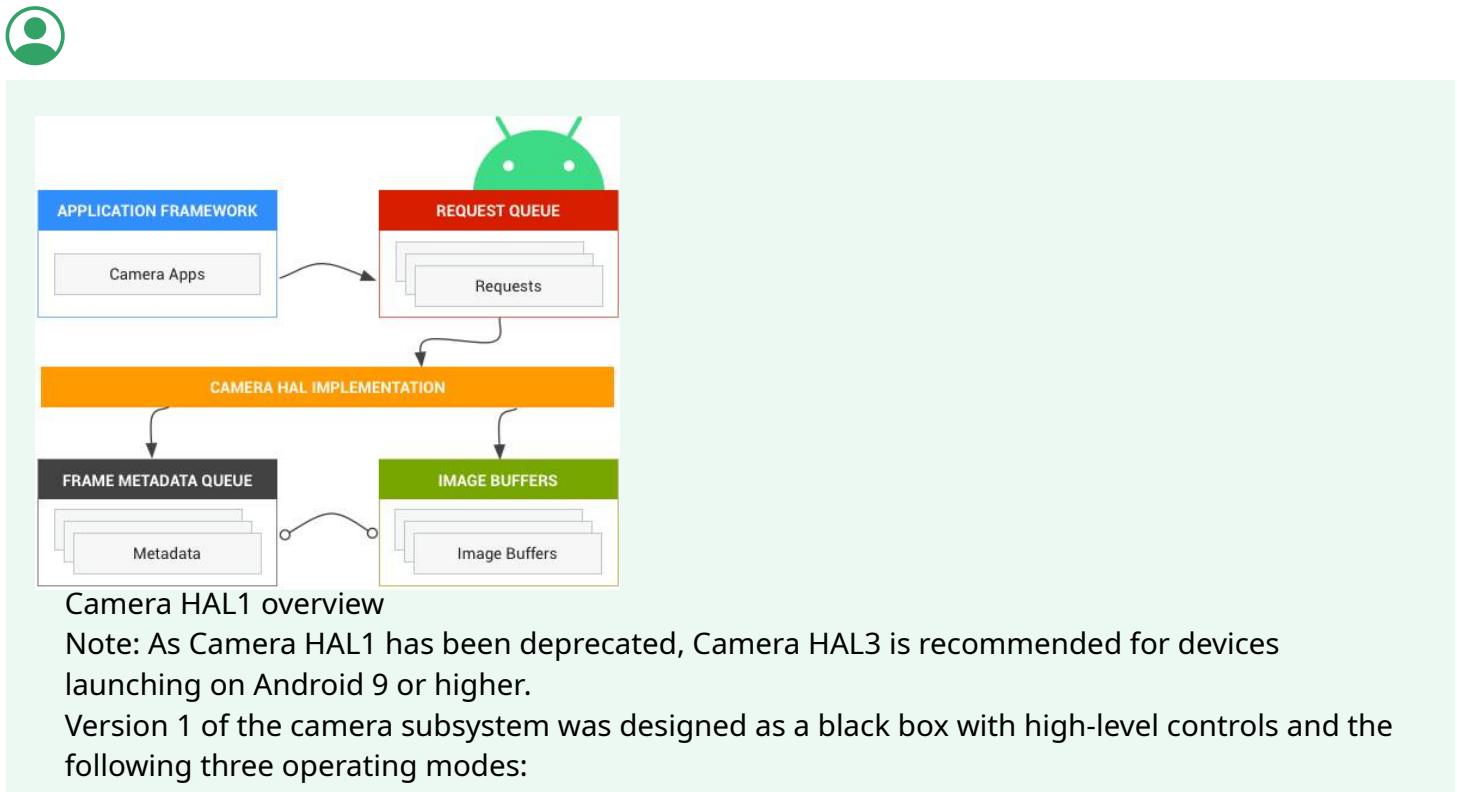
text

App → CameraDevice → CaptureRequest (with Surface targets)
 → CameraFramework prepares request
 → Camera3Device queues request
 → HAL3 receives `buffer_handle_t` (already allocated by framework)
 → HAL uses the buffer to write image data

📌 Summary Table

Buffer Type	Allocated By	Mechanism
App Preview Buffer	App / Framework	Surface / ImageReader / Gralloc
JPEG/YUV Buffer	App / Framework	Gralloc / BufferQueue
Metadata Buffer	HAL (if needed)	<code>malloc</code> / <code>mmap</code> / <code>ion</code>
Internal temp buffer	HAL	platform-dependent

Let me know if you want to trace the actual AOSP source lines for where buffer handles are passed to the HAL or how BufferQueue works.



Preview

Video Record

Still Capture

Each mode has slightly different and overlapping capabilities. This made it hard to implement new features such as burst mode, which falls between two of the operating modes.

Camera block diagram

Figure 2. Camera components

Android 7.0 continues to support camera HAL1 as many devices still rely on it. In addition, the Android camera service supports implementing both HALs (1 and 3), which is useful when you want to support a less-capable front-facing camera with camera HAL1 and a more advanced back-facing camera with camera HAL3.

There is a single camera HAL module (with its own version number), which lists multiple independent camera devices that each have their own version number. Camera module 2 or newer is required to support devices 2 or newer, and such camera modules can have a mix of camera device versions (this is what we mean when we say Android supports implementing both HALs).

Based on the **diagram** you uploaded (Camera HAL1 overview), and the **description**, let's break down the answer to your two key questions:

1 2 1. Number of Queues Used

There are **three logical queues** in the Camera HAL1 architecture:

✓ Request Queue

- **What it holds:** Capture requests from camera apps (e.g., preview, video, still capture).
- **Flow:** Requests go from the application framework → camera HAL implementation.

✓ Frame Metadata Queue

- **What it holds:** Frame-related metadata (timestamp, exposure, white balance, etc.).
- **Flow:** From HAL implementation → framework (so app can know frame details).

✓ Image Buffers

- **What it holds:** Actual pixel/image data captured by the camera.
- **Flow:** From HAL to framework → to app (e.g., via **Surface/ImageReader**).

2. Where Does Buffer Allocation Happen?

◆ App Layer (Framework side)

- **Camera apps or camera service** pre-allocate image buffers via APIs like **ImageReader**, **SurfaceTexture**, etc.
- Buffers are often **GraphicBuffer**, allocated by the **Gralloc** (Graphics Memory Allocator).
- The app hands over these pre-allocated buffers to the camera HAL via the request.

◆ HAL (Driver side)

- HAL **does NOT own the buffers**, it just writes to them.
- HAL expects buffers to be already allocated by the framework and **passed via request queue**.

Special case:

If HAL1 (or HAL3) wants to allocate internal working buffers (e.g., for ISP or post-processing), it can use:

- **malloc**, **mmap**, or
- Android-specific memory allocators like **ION** or **dmabuf**.

Summary

Component	Responsible for Buffer Allocation
Camera App	Initiates allocation via APIs
Framework (CameraService)	Manages and passes to HAL
Camera HAL	Uses passed buffers, may allocate internal ones (e.g., via ION)

If you want, I can also show how this differs in **Camera HAL3** or how queues and buffer management works with **binderized HALs**.



HAL subsystem

bookmark_border

Requests

The app framework issues requests for captured results to the camera subsystem. One request corresponds to one set of results. A request encapsulates all configuration information about the capturing and processing of those results. This includes things such as resolution and pixel format; manual sensor, lens, and flash control; 3A operating modes; RAW to YUV processing

control; and statistics generation. This allows for much more control over the results' output and processing. Multiple requests can be in flight at once, and submitting requests is non-blocking. And the requests are always processed in the order they are received.

Great! You're referring to the **HAL Subsystem – Request flow** in **Camera HAL3**. Let's break down what this means and how it works in simple terms.

HAL Subsystem in Camera HAL3

Requests: What Are They?

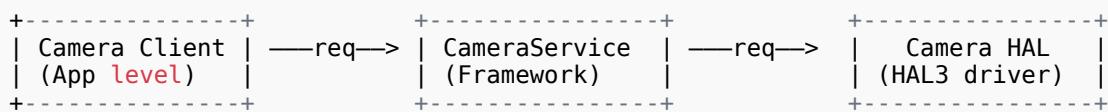
A **Request** in Camera HAL3 is a structured message from the **framework (CameraService)** to the **HAL (vendor implementation)** that tells the camera what to do for a single frame.

Each request includes:

- **What to capture** (e.g. preview, still, video).
- **How to capture** (sensor settings, lens focus, exposure, flash, white balance).
- **Where to store the result** (output buffers for image + metadata).
- **Processing settings** (3A: auto-exposure, auto-focus, auto-white balance, etc.).

Request Lifecycle Overview

```
pgsql
```



Request Internals

A typical **CaptureRequest** contains:

- **camera_metadata**: the controls (e.g. exposure time)
- **output_buffers[]**: where HAL should write image data
- **frame_number**: ID of the frame for tracking
- **settings**: request-specific overrides

Think of it like: "**Frame #105: use these settings, write result here.**"

Multiple In-Flight Requests

- HAL3 allows **pipelining**: several requests can be in progress at the same time.
- This is essential for performance—especially for **streaming preview + capture**.
- But requests must be processed **in-order** by the HAL to maintain predictability.

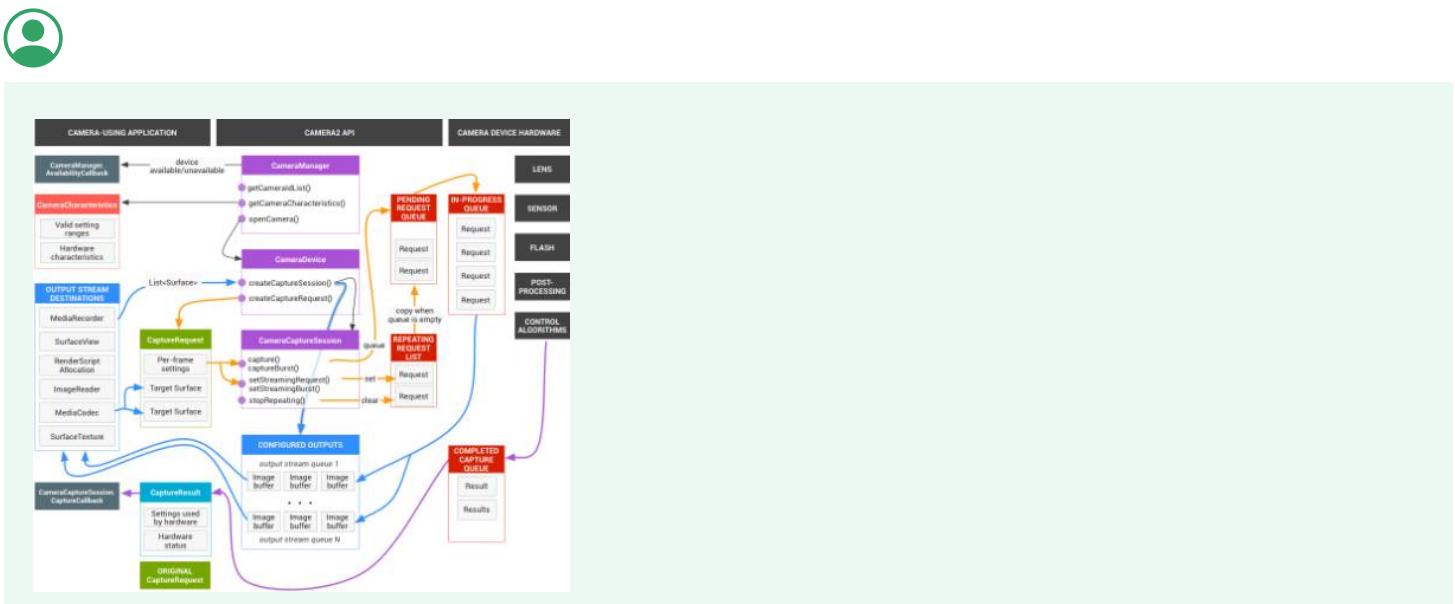
Non-blocking and Ordered

- **Non-blocking submission**: The framework sends the request and moves on; it doesn't wait.
- **In-order processing**: The HAL must process and produce results for the requests in the order received.

Summary Table

Feature	Camera HAL3 (Requests)
Purpose	Full configuration of a single frame
Includes	Settings, output buffers, processing mode
Multiple Requests Allowed?	<input checked="" type="checkbox"/> Yes (in-flight requests supported)
Submission Behavior	<input checked="" type="checkbox"/> Non-blocking
Processing Order	<input checked="" type="checkbox"/> In-order only

Let me know if you'd like a diagram or to explore **how these requests interact with queues, buffers, and the result pipeline** inside HAL.



Great — you've shared the official **Camera2 framework architecture diagram**. Let's answer your core question:

❓ Where does buffer allocation happen in this architecture?

✓ Buffer Allocation Location:

➤ The buffer allocation happens in the **Configured Outputs** section — specifically:

SCSS

- CONFIGURED OUTPUTS → output stream queue 1 .. N → Image **buffer**(s)

These buffers are tied to the **Output Stream Destinations** (e.g., `ImageReader`, `MediaRecorder`, `SurfaceTexture`, etc.).

🔄 Flow of Allocation:

1. App Side:

- You create an `ImageReader` or `SurfaceView`, etc., and provide their **Surface** to `createCaptureSession()`.
- These Surfaces wrap **buffers** internally. For example:
 - `ImageReader` uses `ANativeWindowBuffer`
 - `SurfaceTexture` allocates `GraphicBuffers`
- Buffer allocation is typically done via **Gralloc / ION / DMA-BUF** depending on Android version.

2. `CameraDevice.createCaptureSession()`:

- During session creation, the framework examines each provided surface and **prepares the buffers**.
- It negotiates pixel format, resolution, usage (read/write), and count.

3. Inside HAL:

- HAL receives the buffer handles (shared memory) from framework and **uses them during each request**.
- Buffers are **not allocated inside HAL** but handed over to HAL for each frame.

🔒 What if Secure Memory Is Needed?

- When secure camera is required (e.g. DRM, secure preview):
 - Surface must be marked with secure usage flags.

- Allocation happens using a secure heap (e.g., **ION secure heap, protected memory via Gralloc**).
- HAL recognizes the memory as non-readable by non-secure world.
- Output stays inside **Trusted Execution Environment (TEE)** or **Secure Display pipeline**.

Summary of Buffer Flow:

Stage	What Happens
App	Instantiates SurfaceView , ImageReader , etc.
CameraDevice	createCaptureSession(List<Surface>) passes surfaces
Framework	Allocates GPU-accessible or secure memory buffers
HAL	Receives handles, fills them with sensor/frame data
App	Gets results via CaptureCallback or ImageReader.OnImageAvailableListener

Would you like a visual step-by-step trace through this pipeline for a single capture using **ImageReader + Camera2**?



How is the output from the camera sensor stored and sent back to the app? Explain in simple words

Sure! Here's a **simple explanation** of how the **camera sensor's output** is stored and sent back to your Android app:

1. Camera Sensor Captures Image

- The camera **sensor** (hardware) sees light and captures **raw pixel data** (like a grid of brightness/color values).
- This is **very low-level data**, not ready to show on screen yet.

2. Camera HAL and ISP Process the Data

- The HAL (Hardware Abstraction Layer) and ISP (Image Signal Processor) **clean up and process** the raw sensor data:
 - Remove noise
 - Adjust brightness, color, white balance, focus (3A)
 - Convert from raw to usable format like **YUV** or **JPEG**

3. Output is Put into a Buffer (Memory Box)

- The processed image is stored in a **buffer** — a reserved area in RAM.
- This buffer is not created by HAL. Instead:
 - Your app (via the Camera framework) tells Android what kind of buffer to use.
 - This is done when you give the camera a **Surface** (like `ImageReader`, `SurfaceView`, etc.).

4. Framework Hands HAL the Buffer to Fill

- The Android framework:
 - Takes the buffer from your app's surface.
 - Hands it to the HAL for each camera capture request.
- The HAL **writes the image data** into this buffer.

5. Image Sent Back to Your App

- After HAL fills the buffer:
 - Android notifies your app (e.g. `onImageAvailable()` for `ImageReader`).
- Now your app can:
 - Show the image on screen
 - Save it to storage
 - Do further processing

Example:

If you're using `ImageReader`:

- You set up an `ImageReader` with size & format
- The camera writes each captured frame into `ImageReader`'s internal buffer
- You read it via `imageReader.acquireNextImage()`

If Secure Camera:

- Buffers go into **protected memory**
- App can't read image directly

- Only secure display or secure path can access the data

✓ Summary (Plain Terms):

Step	What Happens
1. Sensor	Takes a photo (raw data)
2. HAL & ISP	Clean up and process the image
3. Buffer	Image is put in a memory box
4. HAL fills it	HAL writes image to the buffer your app gave
5. App gets it	App is notified and reads image from that buffer

Let me know if you want a **visual diagram** for this flow — I can generate it!



which is created first request or session?

Great question! In the Android **Camera2 API**, the **Session** is created **first**, then you create and send **Requests** through that session.

📌 Simple Answer:

👉 **Session is created first**, then you use it to send **capture requests**.

🧠 Why?

The **CameraCaptureSession** represents the actual ongoing connection between the camera device and the app. You **cannot send any capture requests until this session is created**.

✓ Step-by-Step Flow:

Step	What's Created	Why?
1	CameraDevice	Open the physical camera
2	Output Targets	e.g., ImageReader , SurfaceView , etc.
3	CameraCaptureSession	Tells the camera <i>where</i> to send the data
4	CaptureRequest	Tells the camera <i>what</i> to capture (e.g., settings)
5	Send Request	Through the session via capture() or setRepeatingRequest()

Analogy:

Imagine the **Session** is like setting up a video call connection:

-  You first set up the **call session** (CameraCaptureSession)
-  Then you send **messages** or **instructions** (CaptureRequests) during the call

Let me know if you want an illustrated version of this flow.



What's in request?

In simple words:

What's in a Camera Capture Request?

A **CaptureRequest** is a bundle of settings that tells the camera **how** to capture a frame.

It includes:

 Setting	 Meaning
Exposure time	How long to keep the shutter open
Focus settings	Autofocus on/off, focus distance
White balance	Auto/manual color temperature
Sensor sensitivity (ISO)	How bright the image is
Flash mode	On/off/auto
Scene mode	Night, portrait, etc.
Output target(s)	Where to send the image (e.g., ImageReader, SurfaceView)
3A mode	Auto Exposure, Auto Focus, Auto White Balance
Control mode	Manual or auto controls
RAW/YUV/JPEG config	Pixel format and resolution for image output
Metadata tags	Optional data sent along with result

Used How?

- A **CaptureRequest** is created using a **template** (e.g., `TEMPLATE_PREVIEW`, `TEMPLATE_STILL_CAPTURE`)
- Then you customize settings as needed
- Finally, send it via the session using `capture()` or `setRepeatingRequest()`

🎯 Summary:

A **CaptureRequest** is like an instruction sheet that says:
 "Take a photo with these settings, and send the result to this target."

Would you like to see an actual code example too?



Is it created based on whether the output surface or the session?

Great question!

🧩 Is a CaptureRequest created based on output surface or session?

✓ Answer:

A **CaptureRequest** is created **based on the session's output surfaces — not independently.**

💡 Sequence (Simplified):

1. **App provides output surfaces** (e.g. `SurfaceView`, `ImageReader`)
 - These are what the camera will write image data into.
2. **Create CameraCaptureSession using those surfaces**
 - The session links your camera device to output destinations.
3. **Create CaptureRequest from a session-compatible template**
 - e.g., `cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW)`
4. **Add the SAME output surfaces to the request**
 - `requestBuilder.addTarget(surface)`

So:

🔁 **The request depends on the session, and the session depends on the surfaces.**

📌 Why?

This ensures:

- The buffers for camera output are correctly allocated and attached
- The HAL knows where to send the captured frame

Visual Flow:

```
csharp
[SurfaceView/ImageReader]
  ↓
[CameraCaptureSession]
  ↓
[CaptureRequest]
```

Would you like a diagram or code walkthrough?



yeah code

Here's a simple **Android camera2 API preview flow** showing the relationship between **surfaces, session, and request**, with **code comments explaining each step**:

Full Flow: Surface → Session → Request → Preview

```
java
CameraDevice cameraDevice;
CameraCaptureSession captureSession;
CaptureRequest.Builder previewRequestBuilder;
Surface previewSurface; // from a SurfaceView or TextureView

// Step 1: Get the surface (e.g., from TextureView)
SurfaceTexture texture = textureView.getSurfaceTexture();
texture.setDefaultBufferSize(1920, 1080); // Set preview size
previewSurface = new Surface(texture); // Convert to Surface

// Step 2: Create capture request builder
previewRequestBuilder = cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);

// Step 3: Add the target surface to the request
previewRequestBuilder.addTarget(previewSurface);

// Step 4: Create a CameraCaptureSession with the surface
cameraDevice.createCaptureSession(
    Arrays.asList(previewSurface), // surfaces used for output
    new CameraCaptureSession.StateCallback() {
        @Override
        public void onConfigured(@NonNull CameraCaptureSession session) {
            captureSession = session;

            try {
                // Step 5: Build the request
                CaptureRequest previewRequest = previewRequestBuilder.build();

                // Step 6: Start repeating request (continuous preview)
                captureSession.setRepeatingRequest(
                    previewRequest,
                    null, // capture callback

```

```

        null    // handler
    );
} catch (CameraAccessException e) {
    e.printStackTrace();
}
}

@Override
public void onConfigureFailed(@NonNull CameraCaptureSession session) {
    Log.e("Camera", "Session configuration failed");
}
},
null // handler
);

```

Summary:

- **Surface** comes from a **TextureView/SurfaceView** or **ImageReader**
- Session is created **with** the surface
- Request is created **after session setup**
- Request targets the **same surface**
- Only then can camera frames flow from HAL to app

Let me know if you also want code for **still capture** or **ImageReader** (e.g., for saving photos).



types of request

In the Android **Camera2 API**, a **CaptureRequest** represents a single request to capture a frame with specific settings (like exposure, focus, output target, etc.).

There are several **types of requests**, depending on the **template** used during creation:

◆ **CaptureRequest Templates**

You create a request using:

```

java
cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_XXX);

```

Here are the **main types (TEMPLATE_XXX)**:

Template	Use Case	Description
TEMPLATE_PREVIEW	Live preview	Optimized for low-latency preview to screen.

Template	Use Case	Description
TEMPLATE_STILL_CAPTURE	Taking photos	Optimized for still image capture (e.g. auto focus, flash).
TEMPLATE_RECORD	Video recording	Optimized for high frame rate + image quality.
TEMPLATE_VIDEO_SNAPSHOT	Capture during video	Still capture <i>while</i> recording video.
TEMPLATE_MANUAL	Full manual control	Lets app fully control sensor, lens, etc. Only for devices that support LEVEL_FULL .
TEMPLATE_ZERO_SHUTTER_LAG	Fast photo	Used for instant capture with zero shutter lag (ZSL). Often used in burst capture.

Internally, each CaptureRequest contains:

- Output target surface(s)
- Exposure time, ISO
- Focus mode
- Flash mode
- White balance
- 3A settings (auto-exposure, focus, white balance)
- Stream control (e.g., noise reduction, edge mode, etc.)

Example:

```
java

CaptureRequest.Builder request =
    cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_STILL_CAPTURE);

request.addTarget(imageReader.getSurface()); // where image will go
request.set(CaptureRequest.CONTROL_AF_MODE, CaptureRequest.CONTROL_AF_MODE_CONTINUOUS_PICTURE);
request.set(CaptureRequest.CONTROL_AE_MODE, CaptureRequest.CONTROL_AE_MODE_ON_AUTO_FLASH);
```

Would you like to see **CaptureRequest settings list** or **manual mode request examples** too?



Is CameraCaptureSession required or optional?

CameraCaptureSession is **required** in the Android Camera2 API.

Why it is required:

You **must** create a `CameraCaptureSession` before sending any capture requests. It acts as the **connection between the camera device and the output surfaces** (like the preview window or an image reader).

What it does:

- Prepares the camera to send image data to surfaces (like a `SurfaceView`, `TextureView`, `ImageReader`, etc.)
 - Accepts one or more `CaptureRequests` (for preview, still capture, etc.)
 - Optimizes hardware pipelines for the given surface configuration
-

Minimal flow:

```
java

cameraDevice.createCaptureSession(
    Arrays.asList(previewSurface, imageReader.getSurface()),
    new CameraCaptureSession.StateCallback() {
        public void onConfigured(CameraCaptureSession session) {
            // Store the session
            captureSession = session;

            // Create a request
            CaptureRequest.Builder requestBuilder =
                cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW);
            requestBuilder.addTarget(previewSurface);

            // Start repeating preview
            captureSession.setRepeatingRequest(
                requestBuilder.build(), null, backgroundHandler);
        }
    },
    null);
```

 **You cannot call `setRepeatingRequest()` or `capture()` without first creating a `CameraCaptureSession`.**

Summary:

- `CameraCaptureSession` is **mandatory**
- It ties the camera device to the **surfaces** (where output goes)
- Acts as a controller for **request submission and repeating**

Let me know if you want to see a full example from `CameraDevice.open()` to preview.



Why do we need the output Target in both the request and the session?

Excellent question! This gets to the heart of how Android's **Camera2 pipeline** is built for **flexibility, performance, and safety**.

Let's break it down:

Why is the Output Target needed in both:

- `CameraCaptureSession`, and
- Each `CaptureRequest`?

1. Session defines: What is *possible*

The `CameraCaptureSession` is like a contract or pipeline setup phase.

It tells the camera HAL:

"These are all the **surfaces** (targets) I might ever send image data to."

 Internally, this allows:

- Memory buffers to be allocated
- Camera hardware pipelines (ISP paths) to be configured
- Format negotiations (e.g., YUV for preview, JPEG for stills)

 So: **Only surfaces declared in the session can ever be used in a request.**

2. Request defines: What I want to use *right now*

The `CaptureRequest` selects **which** of the session's output targets to use **for that specific frame**.

So if the session registered:

- A preview surface
- A JPEG surface (from `ImageReader`)

Then:

Request	Target(s)	What it does
Preview request	Only preview surface	Live feed to screen
Capture request	Only JPEG surface	High-res photo
ZSL / Dual request	Both surfaces	Show preview <i>and</i> save image

⌚ This gives **per-frame control** over where the image should go.

📦 Analogy:

Think of the `CameraCaptureSession` like opening a **multi-function printer**:

- The printer has:
 - 📄 a tray (preview surface)
 - 📸 a photo scanner (JPEG output)
 - 📡 a fax module (maybe depth or RAW)

When you submit a **job (request)**, you specify:

“Use the scanner only” or “Use tray and fax”.

But first, the printer needs to know all its parts (session) before accepting jobs.

🧠 Summary:

Role	Purpose
<code>CameraCaptureSession</code>	Declares all output surfaces the camera may use
<code>CaptureRequest</code>	Chooses which of those surfaces to use for this capture

You **must** add the output target to both — otherwise the HAL won’t be able to prepare or execute the capture.

Let me know if you want a visual diagram or want to explore how this works internally in `Camera3Device` or `CameraHal`.



In a session, can we have multiple requests?

Yes — absolutely! ✅

In a single `CameraCaptureSession`, you can have **multiple CaptureRequest objects**.

❓ Why?

Because different frames might have:

- Different settings (exposure, flash, ISO, etc.)

- Different output targets
- Different intent (preview, still capture, video record, etc.)

Think of a CameraCaptureSession as a pipeline.

Once it's open, you can:

- Create and send **many requests**, like:
 - One for preview
 - One for autofocus
 - One for a still photo
 - One with a different resolution
- Submit them **individually** (`capture()`)
- Or submit them as a **repeating stream** (`setRepeatingRequest()`)

Types of submission:

Submission Type	What It Does
<code>capture(request)</code>	One-shot capture
<code>setRepeatingRequest(request)</code>	Starts continuous streaming with that request
<code>captureBurst(list_of_requests)</code>	Submit multiple different requests for back-to-back execution
<code>setRepeatingBurst(list_of_requests)</code>	Repeats a cycle of multiple requests (e.g., alternate flash/no-flash)

Example:

```
java

// 1. Open session with SurfaceView and ImageReader
CameraCaptureSession session = ...;

// 2. Create multiple requests
CaptureRequest previewRequest = builder1.build(); // Preview
CaptureRequest stillRequest = builder2.build(); // JPEG capture

// 3. Send them
session.setRepeatingRequest(previewRequest, ...); // Preview loop
session.capture(stillRequest, ...); // Take photo
```

Summary:

✓ Yes — one session supports **many requests**, one at a time or in sequence or in a loop. The session defines **which surfaces are allowed**, and requests define **how and when to use them**. Let me know if you want to see an actual code walkthrough!



HAL and camera subsystem

The camera subsystem includes the implementations for components in the camera pipeline such as the 3A algorithm and processing controls. The camera HAL provides interfaces for you to implement your versions of these components. To maintain cross-platform compatibility between multiple device manufacturers and Image Signal Processor (ISP, or camera sensor) vendors, the camera pipeline model is virtual and does not directly correspond to any real ISP. However, it is similar enough to real processing pipelines so that you can map it to your hardware efficiently. In addition, it is abstract enough to allow for multiple different algorithms and orders of operation without compromising either quality, efficiency, or cross-device compatibility.

The camera pipeline also supports triggers that the app framework can initiate to turn on things such as auto-focus. It also sends notifications back to the app framework, notifying apps of events such as an auto-focus lock or errors.

Great! Here's a simple explanation of the relationship between the **Camera HAL**, the **camera subsystem**, and the **HAL pipeline** you just mentioned:

What is the Camera Subsystem?

The **camera subsystem** is Android's internal camera architecture. It includes:

- The **framework-side logic** (like `CameraManager`, `CameraDevice`, etc.)
- The **3A algorithms**:
 - **Auto-Exposure (AE)**
 - **Auto-White Balance (AWB)**
 - **Auto-Focus (AF)**
- The **image processing control** pipeline (formats, pixel conversion, etc.)
- It defines **how the app talks to the HAL** (not how the HAL implements things).

What is the Camera HAL?

The **Camera HAL** is the **vendor-implemented layer** between Android and your actual hardware.

- It provides functions that Android calls (via `camera3_device_ops`)

- It is responsible for:
 - Talking to the **ISP** (Image Signal Processor)
 - Accessing the **camera sensor**
 - Implementing or hooking in **3A logic**
 - Managing **image buffers** (e.g., for JPEG, YUV, RAW)
 - Handling **triggers** and **events** (like auto-focus lock)
- You write this code as part of your SoC or OEM implementation.

How do they connect?

App / Framework	Camera Subsystem	Camera HAL	Hardware
CameraX / Camera2	Submits CaptureRequests	Implements HAL3 interface	ISP + Sensor
Receives CaptureResult	Controls 3A + pipelines	Talks to driver / firmware	
Calls AF triggers	Sends to HAL for execution	Handles and notifies results	

Why is the camera pipeline "virtual"?

Because Android doesn't assume **how your ISP works**.

It defines a **general model** (like: sensor → demosaic → 3A → format conversion → JPEG/YUV/RAW) — but **you map it** to your hardware however you want.

That means:

- You can reorder steps or implement them differently
- You can even **bypass** parts (e.g., no AF on fixed-focus cameras)
- But the output **must still match the expected Android results**

Benefits of this design:

- Ensures apps run consistently across devices
- Allows flexibility for custom ISPs, 3A logic, etc.
- Makes testing (CTS/VTS) possible by standardizing outputs

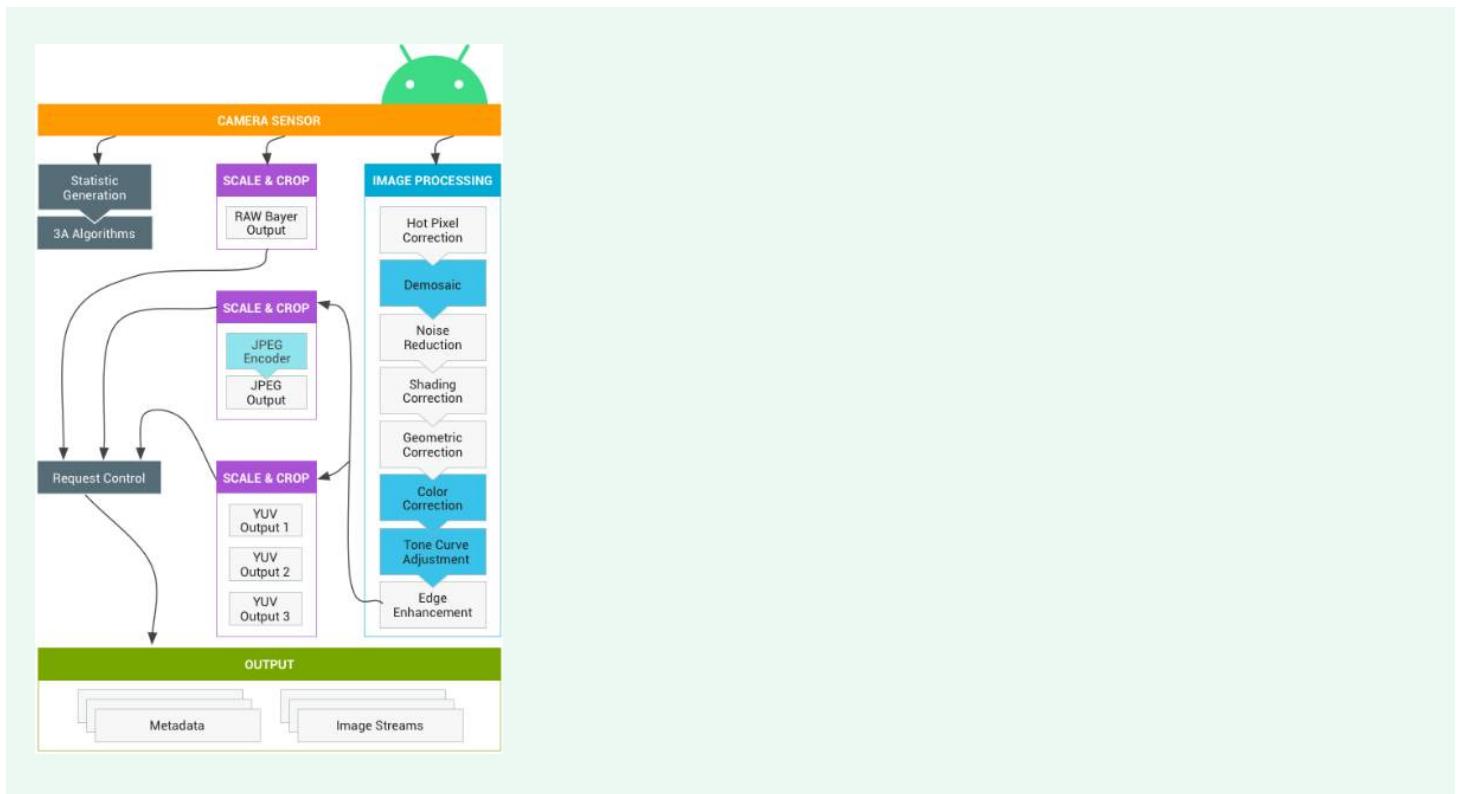
What about triggers and notifications?

These are special **framework-to-HAL events**:

- The app triggers something (like **CONTROL_AF_TRIGGER_START**)

- HAL executes the action (like starting lens movement)
- HAL sends a **callback/notification** back (like AF success/fail)

Let me know if you want a **diagram** or **HAL code structure** too!



This diagram represents **Android's Camera HAL (Hardware Abstraction Layer)** and the **camera processing pipeline** from the sensor to the app. Let's break it down simply:

CAMERA 1. Camera Sensor

- Captures raw light data (like the eye sees light).
- Produces **RAW Bayer** format data.
- Also sends data for **statistics generation** used by 3A algorithms.

CAMERA 2. 3A Algorithms

- **Auto-Focus (AF)**: Makes sure the subject is sharp.
- **Auto-Exposure (AE)**: Controls brightness.
- **Auto-White Balance (AWB)**: Makes colors look natural.

These are continuously calculated using live data from the sensor.

3. Request Control

- From the **framework or app**: Controls what kind of image you want (RAW, JPEG, YUV), with what settings.
 - Tells the pipeline how to process the next frame.
-

4. Image Processing Pipeline

The raw image goes through a series of steps:

- **Hot Pixel Correction**: Fixes sensor defects.
 - **Demosaic**: Converts RAW Bayer to RGB.
 - **Noise Reduction, Shading, Geometric Correction**
 - **Color Correction**: Applies proper color adjustments.
 - **Tone Curve**: Adjusts contrast, brightness.
 - **Edge Enhancement**: Sharpens the image.
-

5. Scale & Crop + Format Conversion

- You can **scale and crop** images in various ways.
 - Three typical output formats:
 - **RAW Bayer**: Raw sensor data (for manual processing).
 - **JPEG**: Compressed image (processed by **JPEG encoder**).
 - **YUV**: Uncompressed, intermediate format (used in video, preview).
-

6. Output

- The app receives:
 - **Metadata**: Info like focus distance, exposure time, ISO, etc.
 - **Image Streams**: Actual image data in requested format (RAW, YUV, JPEG).
-

How It All Connects:

- The **app creates a CaptureRequest**, specifying:

- Output surface (ImageReader, SurfaceView, etc.)
- Format (JPEG/YUV/RAW)
- Settings (ISO, focus, white balance...)
- The HAL uses this to configure the pipeline for each frame.
- The result is sent to the **CaptureSession**, and finally, to the app.

Realtime Loop:

1. Camera sensor captures → data goes to HAL.
2. HAL runs 3A, image processing.
3. Image is formatted and scaled.
4. Data is sent to output surfaces (like the app's preview or ImageReader).

Would you like a real-world analogy or a code sample of how this maps in **Camera2 API**?



Figure 2. Camera pipeline

Please note, some image processing blocks shown in the diagram above are not well-defined in the initial release. The camera pipeline makes the following assumptions:

RAW Bayer output undergoes no processing inside the ISP.

Statistics are generated based off the raw sensor data.

The various processing blocks that convert raw sensor data to YUV are in an arbitrary order.

While multiple scale and crop units are shown, all scaler units share the output region controls (digital zoom). However, each unit may have a different output resolution and pixel format.

Summary of API use

This is a brief summary of the steps for using the Android camera API. See the Startup and expected operation sequence section for a detailed breakdown of these steps, including API calls.

Listen for and enumerate camera devices.

Open device and connect listeners.

Configure outputs for target use case (such as still capture, recording, etc.).

Create request(s) for target use case.

Capture/repeat requests and bursts.

Receive result metadata and image data.

When switching use cases, return to step 3.

HAL operation summary

Asynchronous requests for captures come from the framework.

HAL device must process requests in order. And for each request, produce output result metadata, and one or more output image buffers.

First-in, first-out for requests and results, and for streams referenced by subsequent requests. Timestamps must be identical for all outputs from a given request, so that the framework can match them together if needed.

All capture configuration and state (except for the 3A routines) is encapsulated in the requests and results.

Based on **Figure 2: Camera Pipeline** and the explanation provided, here's a **summary of how the Android camera pipeline and HAL interact**, including answers to your earlier questions:

Why is CameraCaptureSession required?

Yes, **CameraCaptureSession is required**.

It is the component that **manages the flow of capture requests and outputs** between the app and the camera hardware.

- It ties **outputs** (like `SurfaceView`, `ImageReader`, etc.) to the **camera device**.
- It ensures **camera resources are reserved** and **synchronized** for the duration of the session.
- Without a session, you cannot start capturing images or receive image buffers.

Why do we need output targets in both the session and the request?

You need to specify the same outputs in both because:

- The **session** defines the set of possible **valid outputs** (i.e., what surfaces the camera is allowed to write to).
- The **capture request** specifies the **subset of outputs** to be used for that specific capture.

Think of the session as a **menu of available dishes**, and the request as the **order placed**. The order must choose from the menu.

Can a session have multiple capture requests?

Yes.

- A session can handle:
 - **A single repeating request** (e.g. for preview).
 - **One-shot requests** (e.g. take a picture).
 - **Burst requests** (multiple still shots).

- Requests are handled **asynchronously and FIFO (first-in, first-out)**.
-

Camera Pipeline Explained (from image)

Flow Summary:

1. Camera Sensor

- Captures raw data from the scene.

2. Statistics Generation

- From raw sensor data, stats like focus/AE/AWB (3A) are calculated.

3. RAW Bayer Output

- Bypasses ISP processing.

4. Image Processing Path

- Applies processing:
 - Hot Pixel Correction
 - Demosaic
 - Noise Reduction
 - Shading / Geometric / Color Correction
 - Tone Adjustment
 - Edge Enhancement

5. Scale & Crop Units

- Used for different outputs: JPEG and multiple YUV streams.
- **All share zoom/crop window** but can differ in resolution and format.

6. Output Buffers

- **JPEG Output** (after JPEG encoder)
 - **YUV Outputs 1/2/3**
 - **Metadata** is produced for every frame.
-

📌 Notes on HAL Behavior

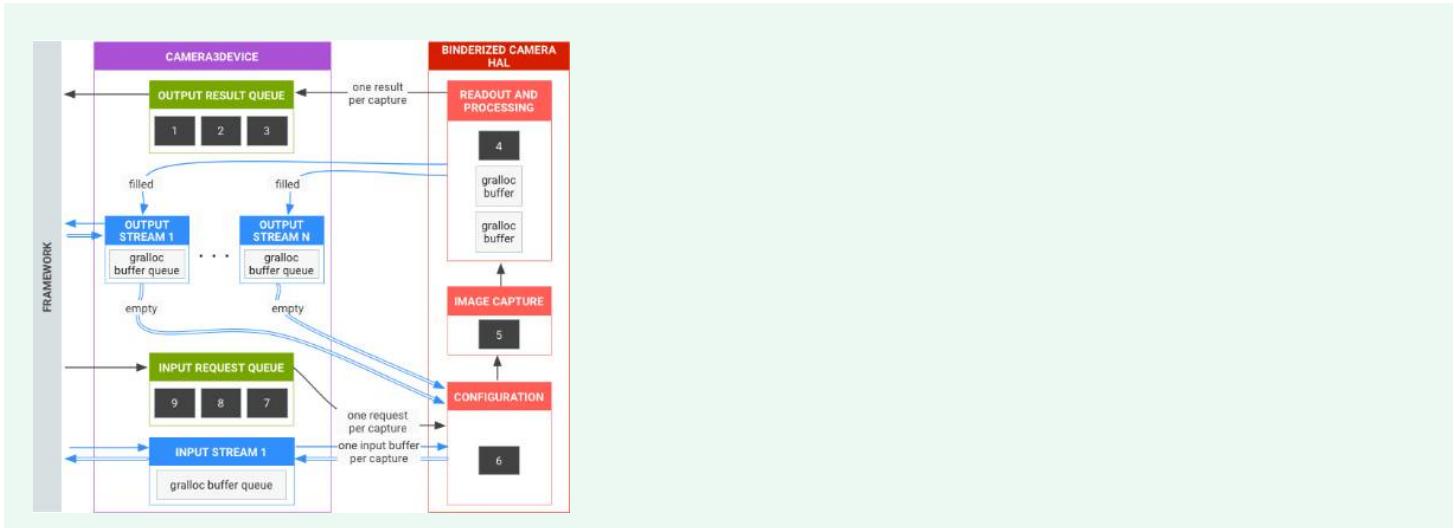
- **Requests from the framework are asynchronous.**
- **HAL processes requests in-order (FIFO).**
- HAL must return:
 - Image data (YUV, JPEG, RAW).

- Metadata (timestamped to match output images).
- **3A routines (auto-focus, exposure, white balance)** are maintained outside of the request pipeline and are often asynchronous.

✓ Summary of API Use Steps:

1. **Detect camera devices** (via `CameraManager`).
2. **Open the camera** (`CameraDevice`) and set a state callback.
3. **Configure outputs** (via `CameraCaptureSession`).
4. **Create capture requests** (`CaptureRequest.Builder`).
5. **Send capture requests**:
 - One-shot: `capture()`
 - Repeating: `setRepeatingRequest()`
6. **Receive output images** (from `ImageReader`) and metadata (`CaptureCallback`).
7. **Switch use-case?**
 - Go back to step 3 (e.g., from preview to recording or high-res capture).

Would you like a visual explanation of how the session, request, and outputs interact in code?



Thanks for sharing the image! Based on this **Camera HAL architecture diagram**, let's break down the **key concepts** and answer your earlier questions step by step:

✓ Q1: Is CameraCaptureSession required or optional?

Answer: Required.

- `CameraCaptureSession` is **essential** in the Android Camera2 API.
- It represents an **active connection** between the app and the camera device for capturing images.
- You **cannot issue capture requests** (`CaptureRequest`) without first creating a `CameraCaptureSession`.

Think of `CameraDevice` as the camera hardware, and `CameraCaptureSession` as the channel through which you tell it what to do.

✓ Q2: Why do we need the output target in both the session and the request?

Answer:

- The **session** is configured with **Surface targets** (e.g., preview display, image reader) via `createCaptureSession()` — this **sets up the pipeline**.
- Each `CaptureRequest` chooses **which of those session outputs to use** and with **what settings** (exposure, focus, etc.).

Why both?

- **Session level** → defines the full pipeline (what outputs can be used).
- **Request level** → chooses **which** of those outputs to actually use **per capture**.

Example: A session might have both preview and JPEG outputs, but a request might only use JPEG.

✓ Q3: In a session, can we have multiple requests?

Answer: Yes.

You can:

1. Use `capture()` → single `CaptureRequest`.
2. Use `captureBurst()` → multiple requests in one batch.
3. Use `setRepeatingRequest()` → continuously repeat a request.
4. Use `setRepeatingBurst()` → continuously repeat a sequence of requests.

The session maintains the output stream setup, and you can change or repeat the capture behavior via different requests.

✓ Explanation of the Diagram You Uploaded

Let's map your image (`camera_hal.png`) to what's happening:

Left Side (Framework/Camera3Device):

- **Input Request Queue:** Holds incoming `CaptureRequests` from framework.
- **Output Streams:** Multiple gralloc buffers where processed frames go.
- **Input Stream:** Used for reprocessing (YUV→JPEG, etc).
- **Output Result Queue:** Holds metadata/results for each request.

Right Side (Binderized HAL):

- **Configuration:** Receives stream info and prepares the HAL.
- **Image Capture:** Captures raw sensor data (Step 5).
- **Readout and Processing:** Converts and processes to gralloc output buffers (Step 4).

Each `capture` uses:

- **One input buffer (optional)** (e.g., reprocessing)
- **One output buffer per stream** (e.g., preview + JPEG)
- **One result metadata packet** (e.g., exposure, timestamp)

Summary Table

Component	Purpose
<code>CameraDevice</code>	Interface to physical camera hardware
<code>CameraCaptureSession</code>	Active session for capturing
<code>CaptureRequest</code>	Per-frame settings + which outputs to target
Output Targets	Declared in session and referred in request
Multiple Requests	Yes, possible via <code>captureBurst</code> or repeating calls

Let me know if you want a code-based walkthrough of these concepts.



confusing

Let's break it down **step by step in a simpler way**, using an analogy 🎥:

🔧 First, Think of a Camera App Like a Factory

- You have a **Camera Device** → like a camera machine.
 - You open a **Session** → like setting up the assembly line.
 - You send **Requests** → like sending orders to the machine.
-

✓ Why Do We Need CameraCaptureSession?

Imagine you want to take pictures or preview the camera:

- You need to tell Android **where** the image should go (e.g., screen for preview or file for JPEG).
- That “where” is given using a **Surface** (like a display window or buffer).
- To **send camera data to that Surface**, you need to create a **CameraCaptureSession**.

📌 So: No session = No pipeline. You can't take pictures or show preview.

✓ Why Do We Need Output Target in Both Session & Request?

Let's say your session says:

"I'll be sending images to Preview and JPEG output."

But each **CaptureRequest** says:

"This frame, I want to send it to Preview only."

"Next frame, I want to send it to JPEG."

So:

Level	What It Says
Session	What outputs are possible (setup time)
Request	What outputs are actually used for this frame (per-frame)

🔄 Session = Setup once, Request = Decide per frame

✓ Can We Have Multiple Requests in a Session?

Yes ✓

You can:

- Send one request → `session.capture(request)`
- Send multiple at once → `session.captureBurst([req1, req2])`
- Repeat a request again and again → `session.setRepeatingRequest(request)`

💡 Think of a session like a **pipeline**, and the requests are like **commands telling it what to do** each time.

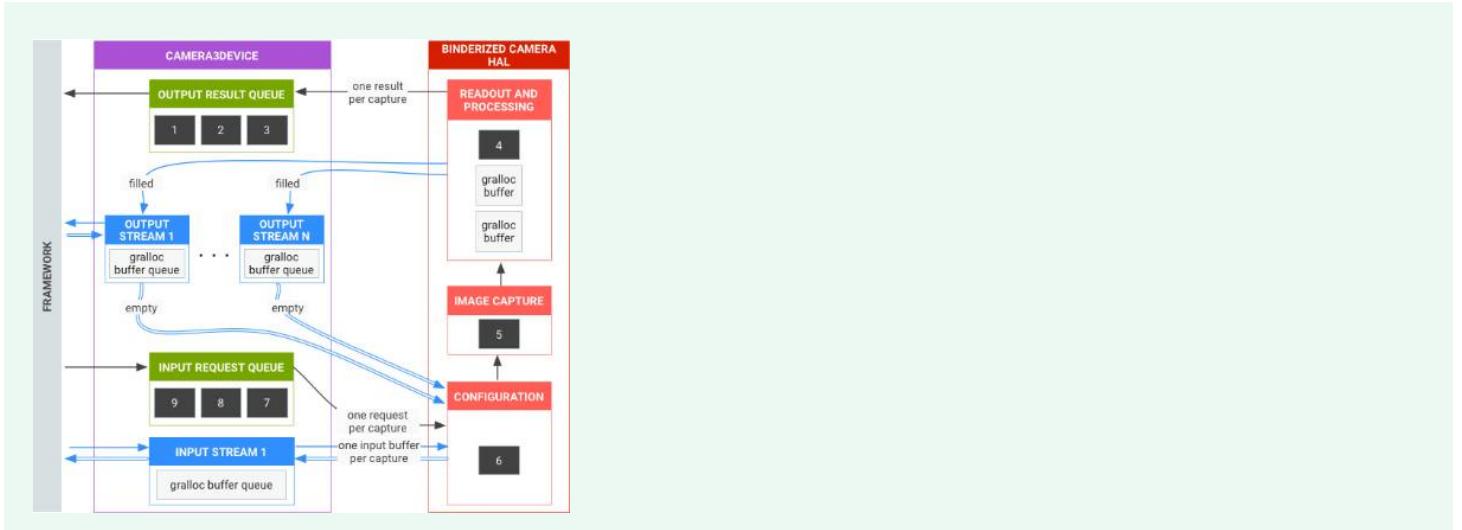
✓ Simple Visual Example

```
less

[CameraDevice]
  |
  V
[CameraCaptureSession] —> Output Surfaces: [Preview Surface, JPEG Surface]
  |
  V
[CaptureRequest 1] → send to Preview only
[CaptureRequest 2] → send to JPEG only
[CaptureRequest 3] → send to both
```

Would you like a **real example code** using Camera2 that shows this in action (preview + capture)? Or a simpler diagram?

Camera HAL-2



This diagram represents the **Android Camera HAL (Hardware Abstraction Layer) pipeline**, specifically how the **Camera3Device** communicates with the **Binderized Camera HAL**. Here's a clear explanation of each part:

Left Side: Framework / Camera3Device

This is the part of the Android framework responsible for managing camera operation.

1. INPUT REQUEST QUEUE

- Holds pending **capture requests** (7, 8, 9).
- Each request includes metadata and optionally an input buffer.

2. INPUT STREAM(S)

- Used if you're reprocessing input images (like effects, HDR, etc.).
- Uses a **gralloc buffer queue** for buffer management.

3. OUTPUT STREAM(S)

- Can have one or more (**OUTPUT STREAM 1 ... OUTPUT STREAM N**).
- Each stream maps to an app-specified destination (e.g., preview, JPEG, YUV).
- Uses **gralloc buffer queues**.

- Buffers are marked **empty** → **filled** as they pass through HAL.

4. OUTPUT RESULT QUEUE

- Stores the result metadata after processing (e.g., focus state, exposure, etc.).
- Delivers **one result per capture** back to the app.

Right Side: Binderized Camera HAL

This is the HAL implementation that interfaces with the physical camera hardware.

5. CONFIGURATION

- Handles setup (e.g., sensor modes, stream formats) via HIDL/AIDL.
- Occurs once during camera startup or when stream settings change.

6. IMAGE CAPTURE

- Triggers the camera sensor to capture an image.
- Uses data from the **INPUT REQUEST QUEUE**.

7. READOUT AND PROCESSING

- Reads out raw sensor data.
- Applies ISP (Image Signal Processor) processing.
- Writes processed data into **gralloc buffers** from the output stream.

Data Flow Summary:

1. **Framework sends a request** (metadata + buffers) → Input Request Queue.
2. Request and empty output buffers are handed to HAL.
3. HAL performs **capture (5)**, then **readout/processing (4)**.
4. HAL fills output buffers, puts them in **Output Streams**.
5. A result is added to **Output Result Queue**.
6. Buffers and metadata are returned to the app.

Key Concepts:

- **gralloc buffers**: Shared memory buffers for graphics (camera uses them to transfer images).

- **One request per capture:** A request maps to exactly one image operation.
- **Binderized HAL:** The HAL operates in a separate process, communicating over Binder IPC.

Let me know if you want a step-by-step walk-through of one capture cycle, or how secure buffers and TEE would integrate into this.



Startup and expected operation sequence

This section contains a detailed explanation of the steps expected when using the camera API. Please see [platform/hardware/interfaces/camera/](#) for HIDL interface definitions.

Enumerate, open camera devices, and create an active session

After initialization, the framework starts listening for any present camera providers that implement the `ICameraProvider` interface. If such provider or providers are present, the framework will try to establish a connection.

The framework enumerates the camera devices via `ICameraProvider::getCameraIdList()`.

The framework instantiates a new `ICameraDevice` by calling the respective `ICameraProvider::getCameraDeviceInterface_VX_X()`.

The framework calls `ICameraDevice::open()` to create a new active capture session `ICameraDeviceSession`.

Use an active camera session

The framework calls `ICameraDeviceSession::configureStreams()` with a list of input/output streams to the HAL device.

The framework requests default settings for some use cases with calls to `ICameraDeviceSession::constructDefaultRequestSettings()`. This may occur at any time after the `ICameraDeviceSession` is created by `ICameraDevice::open`.

The framework constructs and sends the first capture request to the HAL with settings based on one of the sets of default settings, and with at least one output stream that has been registered earlier by the framework. This is sent to the HAL with

`ICameraDeviceSession::processCaptureRequest()`. The HAL must block the return of this call until it is ready for the next request to be sent.

The framework continues to submit requests and calls

`ICameraDeviceSession::constructDefaultRequestSettings()` to get default settings buffers for other use cases as necessary.

When the capture of a request begins (sensor starts exposing for the capture), the HAL calls `ICameraDeviceCallback::notify()` with the SHUTTER message, including the frame number and the timestamp for start of exposure. This notify callback does not have to happen before the first `processCaptureResult()` call for a request, but no results are delivered to an app for a capture until after `notify()` for that capture is called.

After some pipeline delay, the HAL begins to return completed captures to the framework with `ICameraDeviceCallback::processCaptureResult()`. These are returned in the same order as the requests were submitted. Multiple requests can be in flight at once, depending on the pipeline depth of the camera HAL device.

After some time, one of the following will occur:

The framework may stop submitting new requests, wait for the existing captures to complete (all buffers filled, all results returned), and then call `ICameraDeviceSession::configureStreams()` again. This resets the camera hardware and pipeline for a new set of input/output streams. Some streams may be reused from the previous configuration. The framework then continues from the first capture request to the HAL, if at least one registered output stream remains. (Otherwise, `ICameraDeviceSession::configureStreams()` is required first.)

The framework may call `ICameraDeviceSession::close()` to end the camera session. This may be called at any time when no other calls from the framework are active, although the call may block until all in-flight captures have completed (all results returned, all buffers filled). After the `close()` call returns, no more calls to `ICameraDeviceCallback` are allowed from the HAL. Once the `close()` call is underway, the framework may not call any other HAL device functions.

In case of an error or other asynchronous event, the HAL must call

`ICameraDeviceCallback::notify()` with the appropriate error/event message. After returning from a fatal device-wide error notification, the HAL should act as if `close()` had been called on it. However, the HAL must either cancel or complete all outstanding captures before calling `notify()`, so that once `notify()` is called with a fatal error, the framework will not receive further callbacks from the device. Methods besides `close()` should return `-ENODEV` or `NULL` after the `notify()` method returns from a fatal error message.

Great — you're diving into the **Android Camera HAL startup and operation flow**, and the detailed sequence you've provided is core to how Android's **Camera HAL3 (HIDL-based)** interfaces work.

Let me now walk you step-by-step through the **startup and operational flow** as described, along with key context for each stage:

✓ 1. Camera Provider Detection and Device Enumeration

◆ Step: `ICameraProvider` is detected

- **What happens:** The Android framework listens for any services that implement the `ICameraProvider` HIDL interface (e.g., `android.hardware.camera.provider@2.4::ICameraProvider`).
- **When:** Happens during `cameraserver` startup or when `CameraManager` is used in an app.

◆ Step: Get camera device list

cpp

```
ICameraProvider::getCameraIdList()
```

- **Purpose:** Discover all logical/physical camera IDs the HAL exposes (e.g., "0", "1", "0/1" for dual cameras).

✓ 2. Opening the Camera

◆ Step: Get the HAL interface

cpp

```
ICameraProvider::getCameraDeviceInterface_Vx_x()
```

- **What it does:** Returns an instance of **ICameraDevice** HIDL interface for the chosen camera ID.

◆ Step: Open the device

cpp

```
ICameraDevice::open()
```

- **What it does:** Returns a live **ICameraDeviceSession** interface to operate that camera.

✓ 3. Creating Streams and Configuration

◆ Step: Configure input/output streams

cpp

```
ICameraDeviceSession::configureStreams()
```

- **What happens:** Framework tells the HAL which **input/output streams** it will use (e.g., preview, JPEG, YUV).
- **HAL responsibility:** Setup gralloc buffers, ISP, pipeline hardware accordingly.

✓ 4. Getting Default Settings

◆ Step: Construct default request templates

cpp

```
ICameraDeviceSession::constructDefaultRequestSettings(templateId)
```

- **Purpose:** Framework gets default **CameraMetadata** for common use cases like:
 - **TEMPLATE_PREVIEW**
 - **TEMPLATE_STILL_CAPTURE**

- `TEMPLATE_RECORD`

✓ 5. Processing Capture Requests

◆ Step: Submit a request

cpp

```
ICameraDeviceSession::processCaptureRequest()
```

- **Includes:**

- Request settings (`CameraMetadata`)
- Input buffer (optional)
- One or more output buffers

◆ **Note:** This call **blocks** until HAL is ready to accept the next request (ensures backpressure).

✓ 6. Notification of Capture Events

◆ Step: HAL sends shutter event

cpp

```
ICameraDeviceCallback::notify(SHUTTER)
```

- **When:** When the camera sensor starts exposure for that frame.
- **Includes:** Frame number and timestamp.

✓ 7. Delivering Results to Framework

◆ Step: Return capture results

cpp

```
ICameraDeviceCallback::processCaptureResult()
```

- **Contains:**

- Metadata (partial or complete)

- Output buffers (e.g., filled gralloc image data)
- **In-order delivery:** Results are returned **in the same order** as requests.
- **Pipeline depth:** Multiple in-flight requests are allowed based on hardware capacity.

8. Session Reconfiguration or Closure

Option A: Reconfigure streams

- Framework stops requests → waits for all results → calls `configureStreams()` again with a new set.
- Used when app switches from preview to video or capture, or changes resolutions.

Option B: Close the session

cpp

```
ICameraDeviceSession::close()
```

- Can be called **any time** (as long as no other calls are active).
- Blocks until:
 - All in-flight results are returned.
 - All output buffers are filled.

After `close()`:

- HAL must **not call any more callbacks**.
- All HAL functions should return errors or `-ENODEV`.

9. Error Handling via notify()

◆ Step: HAL sends error event

cpp

```
ICameraDeviceCallback::notify(ERROR_*)
```

- Types:
 - `ERROR_DEVICE`
 - `ERROR_REQUEST`
 - `ERROR_RESULT`

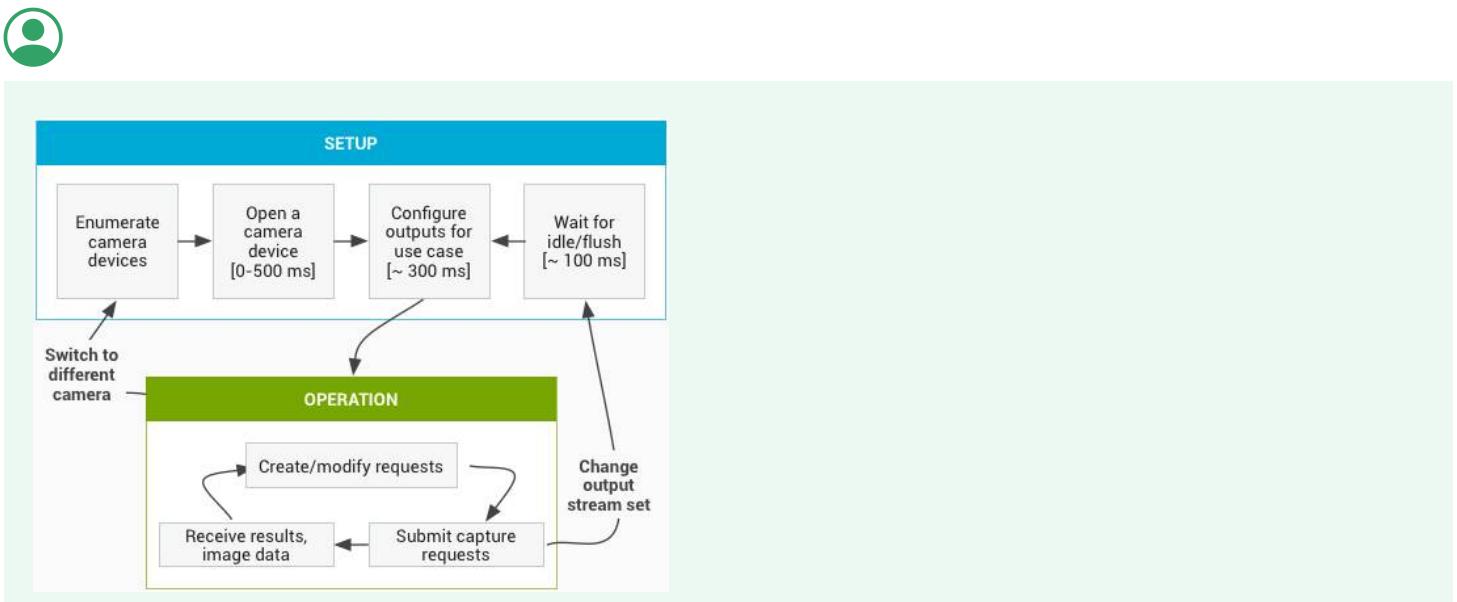
- After a fatal error:

- HAL acts as if `close()` was called.
- Completes or cancels all outstanding requests **before** sending error.
- Returns `-ENODEV` for further framework calls.

Summary Diagram of Expected Flow:



If you'd like, I can also explain how this process looks from the **app's perspective using Camera2 API**, or how to debug issues during any of these stages.



This new diagram is a **simplified flowchart of the camera lifecycle**, showing how **setup and operation phases** are divided in the Android **Camera2 API framework** (which corresponds to the HAL v3/HIDL backend you asked about earlier).

Let's break it down:

◆ SETUP Phase

This phase prepares the camera hardware and its pipeline for capturing images or video. It happens **once** per session (unless reconfiguration is required).

1. Enumerate camera devices

- The framework queries the camera provider to get the list of available camera IDs.
- Internally maps to:

```
cpp  
ICameraProvider::getCameraIdList()
```

2. Open a camera device [0-500 ms]

- Time needed to initialize the selected camera.
- Internally maps to:

```
cpp  
ICameraDevice::open()
```

3. Configure outputs for use case [~300 ms]

- The framework provides output stream definitions (e.g., preview, JPEG) to the HAL.
- HAL configures buffers, ISP paths, etc.
- Internally:

```
cpp  
ICameraDeviceSession::configureStreams()
```

4. Wait for idle/flush [~100 ms]

- Ensures all previously in-flight buffers are drained.
- Required **before changing the stream configuration** (e.g., switch from preview to video).
- Internally uses:

```
cpp
```

```
ICameraDeviceSession::flush() → wait → configureStreams()
```

◆ OPERATION Phase

This is the main image capture phase. It loops continuously until stopped or reconfigured.

5. Create/modify requests

- App prepares **CaptureRequest** with metadata (e.g., exposure, ISO).
- Default settings can be fetched via:

```
cpp
```

```
ICameraDeviceSession::constructDefaultRequestSettings()
```

6. Submit capture requests

- App submits one or more capture requests to the framework.
- Internally maps to:

```
cpp
```

```
ICameraDeviceSession::processCaptureRequest()
```

7. Receive results, image data

- Callback receives:
 - Metadata (e.g., autofocus status)
 - Image buffers (gralloc handles)
- Internally maps to:

```
cpp
```

```
ICameraDeviceCallback::processCaptureResult()
```

Transitions

Change Output Stream Set

- Triggers `configureStreams()` again.
- Requires a wait for idle before doing so.

Switch to a Different Camera

- Calls `close()` on current session and goes back to "Open a camera device" step with a new ID.

Summary

Phase	Tasks	Approx Time
Setup	Enumerate → Open → Configure → Wait	~1 second
Operation	Create → Submit → Receive → (loop)	Real-time

If you're building camera apps or HALs, this flow helps you understand **where delays occur**, and when you're allowed to **reconfigure** or **switch cameras** safely.

Would you like a side-by-side comparison of this app-level flow vs the lower HAL-level API calls?



Hardware levels

Camera devices can implement several hardware levels depending on their capabilities. For more information, see supported hardware level.

Interaction between the app capture request, 3A control, and the processing pipeline
 Depending on the settings in the 3A control block, the camera pipeline ignores some of the parameters in the app's capture request and uses the values provided by the 3A control routines instead. For example, when auto-exposure is active, the exposure time, frame duration, and sensitivity parameters of the sensor are controlled by the platform 3A algorithm, and any app-specified values are ignored. The values chosen for the frame by the 3A routines must be reported in the output metadata. The following table describes the different modes of the 3A control block and the properties that are controlled by these modes. See the `platform/system/media/camera/docs/docs.html` file for definitions of these properties.

Parameter State Properties controlled

`android.control.aeMode` OFF None

ON `android.sensor.exposureTime` `android.sensor.frameDuration` `android.sensor.sensitivity`
`android.lens.aperture` (if supported) `android.lens.filterDensity` (if supported)

ON_AUTO_FLASH Everything is ON, plus android.flash.firingPower, android.flash.firingTime, and android.flash.mode

ON_ALWAYS_FLASH Same as ON_AUTO_FLASH

ON_AUTO_FLASH_RED_EYE Same as ON_AUTO_FLASH

android.control.awbMode OFF None

WHITE_BALANCE_* android.colorCorrection.transform. Platform-specific adjustments if android.colorCorrection.mode is FAST or HIGH_QUALITY.

android.control.afMode OFF None

FOCUS_MODE_* android.lens.focusDistance

android.control.videoStabilization OFF None

ON Can adjust android.scaler.cropRegion to implement video stabilization

android.control.mode OFF AE, AWB, and AF are disabled

AUTO Individual AE, AWB, and AF settings are used

SCENE_MODE_* Can override all parameters listed above. Individual 3A controls are disabled. The controls in the Image Processing block in Figure 2 all operate on a similar principle, and generally each block has three modes:

OFF: This processing block is disabled. The demosaic, color correction, and tone curve adjustment blocks cannot be disabled.

FAST: In this mode, the processing block may not slow down the output frame rate compared to OFF mode, but should otherwise produce the best-quality output it can given that restriction. Typically, this would be used for preview or video recording modes, or burst capture for still images. On some devices, this may be equivalent to OFF mode (no processing can be done without slowing down the frame rate), and on some devices, this may be equivalent to HIGH_QUALITY mode (best quality still does not slow down frame rate).

HIGH_QUALITY: In this mode, the processing block should produce the best quality result possible, slowing down the output frame rate as needed. Typically, this would be used for high-quality still capture. Some blocks include a manual control which can be optionally selected instead of FAST or HIGH_QUALITY. For example, the color correction block supports a color transform matrix, while the tone curve adjustment supports an arbitrary global tone mapping curve.

The maximum frame rate that can be supported by a camera subsystem is a function of many factors:

- Requested resolutions of output image streams
- Availability of binning/skipping modes on the imager
- The bandwidth of the imager interface
- The bandwidth of the various ISP processing blocks

Since these factors can vary greatly between different ISPs and sensors, the camera HAL interface tries to abstract the bandwidth restrictions into as simple model as possible. The model presented has the following characteristics:

The image sensor is always configured to output the smallest resolution possible given the app's requested output stream sizes. The smallest resolution is defined as being at least as large as the largest requested output stream size.

Since any request may use any or all the currently configured output streams, the sensor and

ISP must be configured to support scaling a single capture to all the streams at the same time. JPEG streams act like processed YUV streams for requests for which they are not included; in requests in which they are directly referenced, they act as JPEG streams. The JPEG processor can run concurrently to the rest of the camera pipeline but cannot process more than one capture at a time.

Excellent — you're now exploring **hardware capability levels**, **3A control interaction**, **image processing modes**, and **frame rate constraints** in Android's camera framework. Let me break this down section by section in a clear, structured way:

Camera Hardware Levels (Capability Tiers)

Camera devices declare their **hardware support level** via:

```
pgsql
android.info.supportedHardwareLevel
```

Levels:

Level	Description
LEGACY	Backward compatibility with Camera1 API; limited manual control.
LIMITED	Supports some, but not all, Camera2 features.
FULL	Full manual control (exposure, focus, etc.), high-quality outputs.
LEVEL_3	Adds advanced YUV reprocessing, RAW image capture, and more stream configurations.
EXTERNAL	For USB cameras, webcam-like; limited feature set.

You can check this using:

```
java
cameraCharacteristics.get(CameraCharacteristics.INFO_SUPPORTED_HARDWARE_LEVEL)
```

3A (Auto-Exposure, Auto-White-Balance, Auto-Focus) Behavior

3A controls determine whether certain **capture request fields are respected or overridden**.

Table Summary of 3A Behavior:

Parameter	State	Controlled Properties
<code>android.control.aeMode</code>	OFF	App controls exposure

Parameter	State	Controlled Properties
	ON	AE controls: exposureTime , frameDuration , sensitivity , lens.aperture , filterDensity
	ON_AUTO_FLASH , etc.	Also adds: flash.mode , firingPower , firingTime
android.control.awbMode	OFF	No AWB
	WHITE_BALANCE_*	Overrides: colorCorrection.transform
android.control.afMode	OFF	App sets lens.focusDistance
	FOCUS_MODE_*	AF controls focus
android.control.mode	OFF	All 3A disabled
	AUTO	3A active per settings
	SCENE_MODE_*	Overrides all above with HAL-chosen presets
android.control.videoStabilization	ON	May adjust scaler.cropRegion

📌 **Important:** Even when the app doesn't explicitly set values, **HAL-selected values must be reported in result metadata**, so the app can know what settings were used.

Image Processing Block Modes

These apply to ISP features like:

- **Noise reduction**
- **Edge enhancement**
- **Color correction**
- **Tone mapping**
- **Sharpening**
- **Face beautification** (if supported)

Each block supports:

Mode	Meaning
OFF	Disable the block (not always possible)
FAST	Real-time friendly (e.g., for preview/video)
HIGH_QUALITY	Best quality; slower (e.g., for still capture)

Example:

```
java
```

```
requestBuilder.set(CaptureRequest.NOISE_REDUCTION_MODE,
CaptureRequest.NOISE_REDUCTION_MODE_FAST);
```

Some blocks support **manual override**:

- `colorCorrection.transform` for color correction
- `tonemap.curveRed/Green/Blue` for custom tone mapping

Frame Rate Constraints

The frame rate depends on the **entire camera pipeline**, and the HAL simplifies this with a fixed model:

Key Assumptions:

- **Sensor is set to smallest resolution** that can scale to all active outputs.
- Any capture request can use **any subset** of streams.
- The pipeline must be able to **handle all output streams simultaneously**.

JPEG Note:

- JPEG is treated like a **processed YUV stream** unless it's explicitly requested.
- **JPEG encoder is exclusive**: only one capture can be processed at a time.
- Encoder can **run in parallel** with sensor and ISP.

Summary Chart

Feature Group	HAL Behavior
3A Controls	Auto logic overrides app-set values unless disabled (AE, AWB, AF)
Processing Blocks	Operate in OFF, FAST, or HIGH_QUALITY mode
Frame Rate Limits	Determined by output resolutions + ISP throughput
JPEG Handling	JPEG encoding is serialized; treated as YUV otherwise

Let me know if you want a **real-world example** using Camera2 API where these parameters (e.g., AE/AF/awb) are configured, or a deeper look at how to interpret metadata in the result callback.



What is a stream and why do we need this?

Great question. In Android's **Camera HAL and Camera2 framework**, a **stream** is a fundamental concept that allows the system to understand **where and how the image data will flow**.

What is a Stream?

A **stream** is a **logical connection between the camera and an image consumer**.

It defines:

- **The format** (e.g., `YUV_420_888`, `JPEG`, `RAW`)
- **The size/resolution** (e.g., 1920×1080)
- **The use case** (e.g., preview, video recording, still image capture)

Why Do We Need Streams?

The camera pipeline needs to know **how many outputs**, **what kind of outputs**, and **where the image data should go**. Streams help manage:

1. Buffer Management

- Each stream is backed by a **buffer queue** (e.g., `Gralloc`).
- The HAL knows how many buffers are needed and when they're filled.
- Different streams can share or rotate buffers depending on usage.

2. Parallel Outputs

You can have **multiple simultaneous outputs**:

- Preview to screen (`YUV_420_888` or `SurfaceTexture`)
- JPEG image to file (`JPEG`)
- AI pipeline consuming frames (`YUV` or `RAW`)
- Video encoder receiving buffers (`YUV`)

Each of these needs a **separate stream**, even if they share data internally.

3. Sensor and ISP Configuration

- The HAL must configure the image sensor and ISP to handle all the configured streams **in a single capture operation**.
- Streams define what processing paths are needed (e.g., tone mapping, JPEG compression, scaling).

Stream Lifecycle

1. Configured using:

```
cpp
ICameraDeviceSession::configureStreams()
```

2. Defined by the framework or app:

```
java
new Surface for preview → added as OutputConfiguration
```

3. Used in capture requests:

- One or more output streams per **CaptureRequest**
- Optional input stream for reprocessing (e.g., Zero Shutter Lag)

Real-World Example

Let's say an app wants to:

- Show a **live preview**
- Take a **high-res JPEG photo**
- Record **video**

It will create 3 streams:

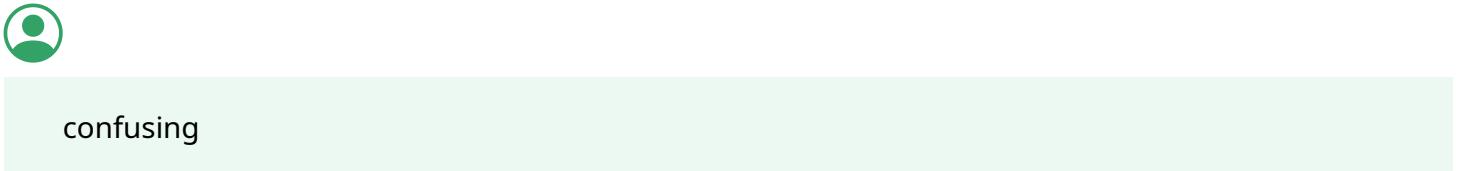
Stream Use	Format	Resolution
Preview	YUV_420_888	1280×720
Still Capture	JPEG	4000×3000
Video	YUV_420_888	1920×1080

These streams are given to the HAL at **configureStreams()**, and the HAL prepares the pipeline accordingly.

Summary

Term	Meaning
Stream	Logical output/input channel (to/from camera) with format, size, and buffer queue
Purpose	Tells HAL what kind of images to produce, how many, and where to send them
Why needed	Enables flexible, parallel image uses: preview, capture, video, reprocessing

Let me know if you want to see how to **set up streams in code (Java or native)** or how streams map to actual hardware paths (sensor → ISP → buffer queue).



No worries — let me explain **what a stream is** and **why it's needed** in a very simple way:

🎥 What is a Camera Stream?

Think of a **camera stream** as a **pipe** through which the camera sends image data to a destination.

Each stream is:

- One pipe
- For one purpose
- With one format (like JPEG, YUV, RAW)
- With one size (like 1920x1080)

🧠 Why do we need streams?

Because apps want to do **different things at the same time** with the camera.

For example:

Action	Stream Created
Show camera preview on screen	● Stream 1: YUV format
Take a photo to save as file	● Stream 2: JPEG format
Record video	● Stream 3: YUV format for encoder

Each one needs a **different format, resolution, and destination** → So we create **separate streams**.

🛠️ How it works

1. The app says:

"I want to show preview on the screen and also save a JPEG image."

2. The camera system sets up two streams:

- One for preview
- One for JPEG photo

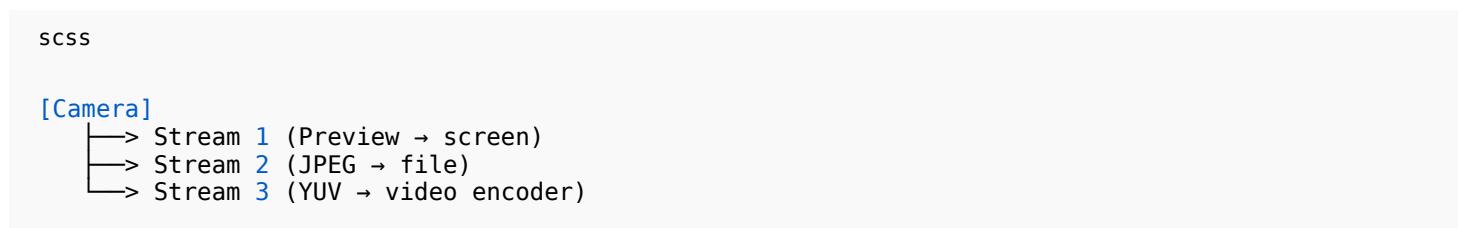
3. Each stream has:

- Format (YUV, JPEG)
- Size (e.g., 1280×720)
- Where to send data (like Surface or ImageReader)

Summary

- A **stream** is just a **data channel** from the camera to where the image goes.
- You need **one stream per output type** (preview, photo, video).
- The camera needs this info so it knows:
 - What to capture
 - How to format it
 - Where to send it

Let me know if you'd like a visual explanation too — I can show a diagram like:





How do these multiple streams work at the same time for a camera?

Great question! Let's break it down simply:

How multiple camera streams work at the same time

Modern Android camera HAL (Hardware Abstraction Layer) allows the camera hardware to **output multiple streams simultaneously** by splitting and formatting the sensor data differently **in parallel**.

Think of the camera sensor like a factory conveyor belt:

1. Sensor captures a full-resolution image
2. Camera HAL creates **multiple parallel pipelines (streams)**:
 - One to shrink and send data to the **screen** (preview)
 - One to compress and send to **JPEG encoder** (photo)
 - One to encode and send to **video encoder** (recording)

The hardware is smart enough to **split and process** the same raw data for different uses at the same time.

What enables this?

1. Stream configuration:

In the `CameraDevice.createCaptureSession()`, the app tells the HAL:

"I want these 3 outputs: a 720p YUV preview, a 1080p JPEG, and a 1080p video stream"

2. HAL allocates buffers and routing logic:

HAL sets up buffers and routes data to:

- GPU/display for preview
- Encoder for video
- JPEG compressor for still image

3. Each stream is attached to a Surface:

The HAL sends image data into:

- `SurfaceView` or `TextureView` → for preview
- `ImageReader` → for image capture
- `MediaRecorder` → for video recording

Internally, the HAL uses:

- **ISP (Image Signal Processor)**: Splits, scales, formats the image
- **Multiple DMA engines**: Push data into different memory streams
- **Buffer queues**: Keeps stream data independent (preview vs capture)

✓ Example: What happens during a photo capture while preview is running

Step	Action
1	Preview stream is sending 720p frames continuously
2	User presses shutter
3	HAL saves one full-resolution frame for JPEG stream
4	Preview continues unaffected
5	JPEG frame is compressed and written to file

🔧 Summary

- Multiple streams = Multiple parallel outputs from the same camera
- HAL uses hardware (ISP, DMA) to split and process data
- Each stream has its own **format, size, and destination**
- All configured at once using a `CameraCaptureSession`

Let me know if you'd like a diagram to go along with this explanation!



Is it a stream for a one request, and when does the stream come to Android?

Great follow-up! Let's clarify everything step by step:

✓ Is a stream for one request only?

✗ No — a stream is not limited to just one request.

- Instead:
- A **stream is configured once** (during `configureStreams()`).
- Then it can be reused in **many capture requests**.

Example:

- You configure a **preview stream** (say, 1280×720 YUV).
- Then you send 100 **captureRequests** — **each one** uses that same stream.

- Only when you want to **change resolution, format, or add/remove outputs**, you must reconfigure the streams.

How stream and request interact

- **Stream** = What the camera HAL can output (format + resolution + target buffer).
- **Request** = What you want the camera to capture for **one frame**.

So:

- Requests **refer to** one or more existing streams.
- The **same stream** (like preview) can be used in every request.
- You can include multiple streams in a single request too.

When does a stream come into Android?

Streams are **created and sent to the HAL during session setup**.

Full lifecycle:

1. App creates **output targets** (like a **Surface** for preview):

```
java
Surface previewSurface = surfaceView.getHolder().getSurface();
```

2. App creates a **capture session** with these surfaces:

```
java
cameraDevice.createCaptureSession(
    Arrays.asList(previewSurface, imageReader.getSurface(), videoSurface),
    sessionCallback, handler);
```

3. Internally, Android framework calls:

```
cpp
ICameraDeviceSession::configureStreams()
```

- This tells the HAL:

"Here are the streams I want: one YUV for preview, one JPEG for still capture."

4. HAL sets up internal pipelines (e.g., buffer queues, ISP paths).

✓ From then on:

- Every `CaptureRequest` refers to **existing streams**.
- The app sends requests using:

```
java
```

```
captureRequestBuilder.addTarget(previewSurface);
cameraCaptureSession.setRepeatingRequest(captureRequestBuilder.build(), ...);
```

🧠 Summary

Concept	Meaning
Stream	A persistent data output channel (format + size + target). Set once.
Request	A one-time command: "Capture a frame using these settings into these streams."
When do streams come?	During session setup (via <code>configureStreams()</code>), before any capture happens.

Let me know if you'd like a code-level example or diagram for this!



so when stream concept came into android

Great question!

⌚ When did the concept of streams come into Android?

✓ Streams were introduced with the

Camera2 API in Android 5.0 (Lollipop).

🔄 Camera1 vs Camera2

Feature	Camera1 (Legacy)	Camera2 (Modern, since Android 5.0)
API Type	Imperative	Declarative, Request-based
Output Handling	One output at a time	Multiple simultaneous streams

Feature	Camera1 (Legacy)	Camera2 (Modern, since Android 5.0)
Control	Limited	Full manual control
Streams	✗ No concept	✓ Introduced and essential

📦 In Camera2, the stream concept appears in:

- `CameraCaptureSession`
- `CaptureRequest`
- HAL-level `configureStreams()` call
- `camera_metadata` keys like `SCALER_AVAILABLE_STREAM_CONFIGURATIONS`

📸 Why were streams introduced?

Because **modern use-cases require multiple outputs simultaneously**, e.g.:

Use Case	Stream Type
Preview on screen	YUV / SurfaceView
Record to file	H.264 / MediaRecorder
Capture high-quality photo	JPEG / ImageReader
Real-time analysis (ML, Face detection)	YUV / ImageReader

To support this, the camera HAL needed a **persistent, reusable** way to configure where frames go — hence the **stream** abstraction.

🧠 Summary

- ✓ **Stream** concept was introduced in **Android 5.0 (Lollipop)** with **Camera2 API**.
- It allows multiple simultaneous outputs like preview + capture + analysis.
- It is configured **once** per session, reused across many requests.
- HAL-side stream support was formalized via `ICameraDeviceSession::configureStreams()`.

Would you like a visual timeline or HAL-level detail on how `configureStreams()` works internally?



Great — let's go step by step and explain **where** stream configuration happens in Android's camera framework.

✓ Summary First:

🔧 Stream configuration happens in:

- **Framework:** via `CameraCaptureSession` setup
- **HAL Interface:** via `ICameraDeviceSession::configureStreams()`

➡ Full Flow: Where Stream Configuration Happens

1. App Side (Java/Kotlin)

Your app calls:

```
java
cameraDevice.createCaptureSession(List<Surface>, ...);
```

This provides:

- List of **Surfaces** (for preview, JPEG, etc.)
- Each **Surface** maps to one **Stream**.

📦 2. Camera Framework (C++)

Framework calls:

```
cpp
ICameraDeviceSession::configureStreams()
```

- This is in `hardware/interfaces/camera/device/3.4/ICameraDeviceSession.hal` (or higher version).
- The input is a **StreamConfiguration** structure:

```
cpp
struct StreamConfiguration {
    vec<Stream> streams;
    StreamConfigurationMode operationMode;
};
```

Each **Stream** contains:

- Width, Height
- Format (e.g., YUV, JPEG)
- Usage flags
- Stream type (OUTPUT or BIDIRECTIONAL)

3. HAL Side (Vendor Implementation)

In vendor code:

cpp

```
Return<Status> CameraDeviceSession::configureStreams(const StreamConfiguration& config)
```

- This is where **buffers get allocated**
- Internal hardware pipelines (ISP, scaler, rotator) are set up
- HAL might decide to reuse existing stream pipelines if config is compatible

Internals:

- For every stream, the HAL sets up buffer queues using **gralloc/ion/ashmem**.
- It maps formats like **HAL_PIXEL_FORMAT_IMPLEMENTATION_DEFINED**, **YUV_420_888**, etc.
- It stores the stream info internally (stream ID, buffer pool, resolution).

4. Requests Use Configured Streams

After **configureStreams()**:

- The app can submit **CaptureRequests** targeting any of the streams.

java

```
builder.addTarget(surface); // Targets a stream
```

At HAL level:

cpp

```
processCaptureRequest(request);
```

This uses previously configured streams — **no need to reconfigure** each time.

TL;DR

Step	Layer	What Happens
1	App	App passes surfaces to createCaptureSession()
2	Framework	Calls configureStreams() with stream info
3	HAL	Sets up buffers and pipelines per stream
4	Request	Streams are used for each capture request

Let me know if you want a diagram showing this flow visually!



3A modes and state transition

bookmark_border

This page describes the 3A modes and state machines for an Android device. The camera HAL interface, which defines the state machines at a high level, lets the HAL implementation and Android framework communicate the current 3A states and trigger 3A events. The HAL implementation is responsible for the 3A algorithms that control the 3A mode settings and state transitions.

When the device is opened, all the individual 3A states must be STATE_INACTIVE. Stream configuration does not reset 3A. For example, locked focus must be maintained across the configure() call.

Triggering a 3A action involves simply setting the relevant trigger entry in the settings for the next request to indicate start of trigger. For example, the trigger for starting an autofocus scan is setting the entry ANDROID_CONTROL_AF_TRIGGER to ANDROID_CONTROL_AF_TRIGGER_START for one request; and cancelling an autofocus scan is triggered by setting ANDROID_CONTROL_AF_TRIGGER to ANDROID_CONTROL_AF_TRIGGER_CANCEL. Otherwise, the entry will not exist or be set to ANDROID_CONTROL_AF_TRIGGER_IDLE. Each request with a trigger entry set to a non-IDLE value will be treated as an independent triggering event.

At the top level, 3A is controlled by the ANDROID_CONTROL_MODE setting. It selects between no 3A (ANDROID_CONTROL_MODE_OFF), normal AUTO mode (ANDROID_CONTROL_MODE_AUTO), and using the scene mode setting (ANDROID_CONTROL_USE_SCENE_MODE):

In OFF mode, each of the individual autofocus (AF), autoexposure (AE), and autowhitebalance

(AWB) modes are effectively OFF, and none of the capture controls may be overridden by the 3A routines.

In AUTO mode, AF, AE, and AWB modes all run their own independent algorithms, and have their own mode, state, and trigger metadata entries, as listed in the next section.

In USE_SCENE_MODE, the value of the ANDROID_CONTROL_SCENE_MODE entry must be used to determine the behavior of 3A routines. In SCENE_MODEs other than FACE_PRIORITY, the HAL must override the values of ANDROID_CONTROL_AE/AWB/AF_MODE to be the mode it prefers for the selected SCENE_MODE. For example, the HAL may prefer SCENE_MODE_NIGHT to use CONTINUOUS_FOCUS AF mode. Any user selection of AE/AWB/AF_MODE when scene must be ignored for these scene modes.

For SCENE_MODE_FACE_PRIORITY, the AE/AWB/AFMODE controls work as in ANDROID_CONTROL_MODE_AUTO, but the 3A routines must bias toward metering and focusing on any detected faces in the scene.

Autofocus settings and result entries

Main metadata entries

ANDROID_CONTROL_AF_MODE Control for selecting the current autofocus mode. Set by the framework in the request settings.

AF_MODE_OFF AF is disabled; the framework/app directly controls lens position.

AF_MODE_AUTO Single-sweep autofocus. No lens movement unless AF is triggered.

AF_MODE_MACRO Single-sweep up-close autofocus. No lens movement unless AF is triggered

AF_MODE_CONTINUOUS_VIDEO Smooth continuous focusing, for recording video.

Triggering immediately locks focus in current position. Canceling resumes continuous focusing.

AF_MODE_CONTINUOUS_PICTURE Fast continuous focusing, for zero-shutter-lag still capture.

Triggering locks focus once currently active sweep concludes. Canceling resumes continuous focusing.

AF_MODE_EDOF Advanced extended depth of field focusing. There is no autofocus scan, so triggering one or canceling one has no effect. Images are focused automatically by the HAL.

ANDROID_CONTROL_AF_STATE Dynamic metadata describing the current AF algorithm state, reported by the HAL in the result metadata.

AF_STATE_INACTIVE No focusing has been done, or algorithm was reset. Lens is not moving. Always the state for MODE_OFF or MODE_EDOF. When the device is opened, it must start in this state.

AF_STATE_PASSIVE_SCAN A continuous focus algorithm is currently scanning for good focus. The lens is moving.

AF_STATE_PASSIVE_FOCUSED A continuous focus algorithm believes it is well focused. The lens is not moving. The HAL may spontaneously leave this state.

AF_STATE_PASSIVE_UNFOCUSED A continuous focus algorithm believes it is not well focused. The lens is not moving. The HAL may spontaneously leave this state.

AF_STATE_ACTIVE_SCAN A scan triggered by the user is underway.

AF_STATE_FOCUSED_LOCKED The AF algorithm believes it is focused. The lens is not moving.

AF_STATE_NOT_FOCUSED_LOCKED The AF algorithm has been unable to focus. The lens is not moving.

ANDROID_CONTROL_AF_TRIGGER Control for starting an autofocus scan, the meaning of which depends on mode and state. Set by the framework in the request settings.

AF_TRIGGER_IDLE No current trigger.

AF_TRIGGER_START Trigger start of AF scan. Effect depends on mode and state.

AF_TRIGGER_CANCEL Cancel current AF scan if any, and reset algorithm to default.

Additional metadata entries

ANDROID_CONTROL_AF_REGIONS Control for selecting the regions of the field of view (FOV) that should be used to determine good focus. This applies to all AF modes that scan for focus. Set by the framework in the request settings.

Autoexposure settings and result entries

Main metadata entries

ANDROID_CONTROL_AE_MODE Control for selecting the current autoexposure mode. Set by the framework in the request settings.

AE_MODE_OFF Autoexposure is disabled; the user controls exposure, gain, frame duration, and flash.

AE_MODE_ON Standard autoexposure, with flash control disabled. User may set flash to fire or to torch mode.

AE_MODE_ON_AUTO_FLASH Standard autoexposure, with flash on at HAL's discretion for precapture and still capture. User control of flash disabled.

AE_MODE_ON_ALWAYS_FLASH Standard autoexposure, with flash always fired for capture, and at HAL's discretion for precapture. User control of flash disabled.

AE_MODE_ON_AUTO_FLASH_REDEYE Standard autoexposure, with flash on at HAL's discretion for precapture and still capture. Use a flash burst at end of precapture sequence to reduce redeye in the final picture. User control of flash disabled.

AE_MODE_ON_LOW_LIGHT_BOOST_BRIGHTNESS_PRIORITY Low Light Boost automatically adjusts the brightness of the Preview stream in low-light conditions. This can result in increased motion blur and noise in low-light conditions. Device manufacturers must ensure the frame rate doesn't go below 10 fps.

ANDROID_CONTROL_AE_STATE Dynamic metadata describing the current AE algorithm state, reported by the HAL in the result metadata.

AE_STATE_INACTIVE Initial AE state after mode switch. When the device is opened, it must start in this state.

AE_STATE_SEARCHING AE is not converged to a good value and is adjusting exposure parameters.

AE_STATE_CONVERGED AE has found good exposure values for the current scene, and the exposure parameters are not changing. HAL may spontaneously leave this state to search for a better solution.

AE_STATE_LOCKED AE has been locked with the AE_LOCK control. Exposure values are not changing.

AE_STATE_FLASH_REQUIRED The HAL has converged exposure but believes flash is required for a sufficiently bright picture. Used for determining if a zero-shutter-lag frame can be used.

AE_STATE_PRECAPTURE The HAL is in the middle of a precapture sequence. Depending on AE mode, this mode may involve firing the flash for metering or a burst of flash pulses for redeye reduction.

ANDROID_CONTROL_AE_PRECAPTURE_TRIGGER Control for starting a metering sequence before capturing a high-quality image. Set by the framework in the request settings.

PRECAPTURE_TRIGGER_IDLE No current trigger.

PRECAPTURE_TRIGGER_START Start a precapture sequence. The HAL should use the subsequent requests to measure good exposure/white balance for an upcoming high-

resolution capture.

Additional metadata entries

ANDROID_CONTROL_AE_LOCK Control for locking AE controls to their current values.

ANDROID_CONTROL_AE_EXPOSURE_COMPENSATION Control for adjusting AE algorithm target brightness point.

ANDROID_CONTROL_AE_TARGET_FPS_RANGE Control for selecting the target frame rate range for the AE algorithm. The AE routine cannot change the frame rate to be outside these bounds.

ANDROID_CONTROL_AE_REGIONS Control for selecting the regions of the FOV that should be used to determine good exposure levels. This applies to all AE modes besides OFF.

Autowhitebalance settings and result entries

Main metadata entries

ANDROID_CONTROL_AWB_MODE Control for selecting the current white-balance mode.

AWB_MODE_OFF Automatic whitebalance is disabled. User controls color matrix.

AWB_MODE_AUTO Automatic white balance is enabled; 3A controls color transform, possibly using more complex transforms than a simple matrix.

AWB_MODE_INCANDESCENT Fixed white balance settings good for indoor incandescent (tungsten) lighting, roughly 2700K.

AWB_MODE_FLUORESCENT Fixed white balance settings good for fluorescent lighting, roughly 5000K.

AWB_MODE_WARM_FLUORESCENT Fixed white balance settings good for fluorescent lighting, roughly 3000K.

AWB_MODE_DAYLIGHT Fixed white balance settings good for daylight, roughly 5500K.

AWB_MODE_CLOUDY_DAYLIGHT Fixed white balance settings good for clouded daylight, roughly 6500K.

AWB_MODE_TWILIGHT Fixed white balance settings good for near-sunset/sunrise, roughly 15000K.

AWB_MODE_SHADE Fixed white balance settings good for areas indirectly lit by the sun, roughly 7500K.

ANDROID_CONTROL_AWB_STATE Dynamic metadata describing the current AWB algorithm state, reported by the HAL in the result metadata.

AWB_STATE_INACTIVE Initial AWB state after mode switch. When the device is opened, it must start in this state.

AWB_STATE_SEARCHING AWB is not converged to a good value and is changing color adjustment parameters.

AWB_STATE_CONVERGED AWB has found good color adjustment values for the current scene, and the parameters are not changing. HAL may spontaneously leave this state to search for a better solution.

AWB_STATE_LOCKED AWB has been locked with the AWB_LOCK control. Color adjustment values are not changing.

Additional metadata entries

ANDROID_CONTROL_AWB_LOCK Control for locking AWB color adjustments to their current values.

ANDROID_CONTROL_AWB_REGIONS Control for selecting the regions of the FOV that should be used to determine good color balance. This applies only to autowhitebalance mode.

General state machine transition notes

Switching between AF, AE, or AWB modes always resets the algorithm's state to INACTIVE. Similarly, switching between CONTROL_MODE or CONTROL_SCENE_MODE if CONTROL_MODE == USE_SCENE_MODE resets all the algorithm states to INACTIVE.

The tables below are per-mode.

AF state machines

mode = AF_MODE_OFF or AF_MODE_EDOF

State	Transformation cause	New state	Notes
-------	----------------------	-----------	-------

INACTIVE AF is disabled

mode = AF_MODE_AUTO or AF_MODE_MACRO

State	Transformation cause	New state	Notes
-------	----------------------	-----------	-------

INACTIVE AF_TRIGGER ACTIVE_SCAN

Start AF sweep

Lens now moving

ACTIVE_SCAN AF sweep done FOCUSED_LOCKED

If AF successful

Lens now locked

ACTIVE_SCAN AF sweep done NOT_FOCUSED_LOCKED

If AF successful

Lens now locked

ACTIVE_SCAN AF_CANCEL INACTIVE

Cancel/reset AF

Lens now locked

FOCUSSED_LOCKED AF_CANCEL INACTIVE Cancel/reset AF

FOCUSSED_LOCKED AF_TRIGGER ACTIVE_SCAN

Start new sweep

Lens now moving

NOT_FOCUSED_LOCKED AF_CANCEL INACTIVE Cancel/reset AF

NOT_FOCUSED_LOCKED AF_TRIGGER ACTIVE_SCAN

Start new sweep

Lens now moving

All states Mode change INACTIVE

mode = AF_MODE_CONTINUOUS_VIDEO

State Transformation cause New state Notes
 INACTIVE HAL initiates new scan PASSIVE_SCAN
 Start AF sweep

Lens now moving

INACTIVE AF_TRIGGER NOT_FOCUSED_LOCKED
 AF state query

Lens now locked

PASSIVE_SCAN HAL completes current scan PASSIVE_FOCUSED
 End AF scan

Lens now locked

PASSIVE_SCAN AF_TRIGGER FOCUSED_LOCKED
 Immediate transformation if focus is good

Lens now locked

PASSIVE_SCAN AF_TRIGGER NOT_FOCUSED_LOCKED
 Immediate transformation if focus is bad

Lens now locked

PASSIVE_SCAN AF_CANCEL INACTIVE
 Reset lens position

Lens now locked

PASSIVE_FOCUSED HAL initiates new scan PASSIVE_SCAN
 Start AF scan

Lens now moving

PASSIVE_FOCUSED AF_TRIGGER FOCUSED_LOCKED
 Immediate transformation if focus is good

Lens now locked

PASSIVE_FOCUSED AF_TRIGGER NOT_FOCUSED_LOCKED Immediate transformation if
 focus is bad

Lens now locked

FOCUSED_LOCKED AF_TRIGGER FOCUSED_LOCKED No effect

FOCUSED_LOCKED AF_CANCEL INACTIVE Restart AF scan
 NOT_FOCUSED_LOCKED AF_TRIGGER NOT_FOCUSED_LOCKED No effect
 NOT_FOCUSED_LOCKED AF_CANCEL INACTIVE Restart AF scan
 mode = AF_MODE_CONTINUOUS_PICTURE
 State Transformation cause New state Notes
 INACTIVE HAL initiates new scan PASSIVE_SCAN
 Start AF scan

Lens now moving

INACTIVE AF_TRIGGER NOT_FOCUSED_LOCKED
 AF state query

Lens now locked

PASSIVE_SCAN HAL completes current scan PASSIVE_FOCUSED End AF scan
 Lens now locked

PASSIVE_SCAN AF_TRIGGER FOCUSED_LOCKED
 Eventual transformation once focus good

Lens now locked

PASSIVE_SCAN AF_TRIGGER NOT_FOCUSED_LOCKED
 Eventual transformation if cannot focus

Lens now locked

PASSIVE_SCAN AF_CANCEL INACTIVE
 Reset lens position

Lens now locked

PASSIVE_FOCUSED HAL initiates new scan PASSIVE_SCAN
 Start AF scan

Lens now moving

PASSIVE_FOCUSED AF_TRIGGER FOCUSED_LOCKED
 Immediate transformation if focus is good

Lens now locked

PASSIVE_FOCUSED AF_TRIGGER NOT_FOCUSED_LOCKED
 Immediate transformation if focus is bad

Lens now locked

FOCUSED_LOCKED AF_TRIGGER FOCUSED_LOCKED No effect
 FOCUSED_LOCKED AF_CANCEL INACTIVE Restart AF scan
 NOT_FOCUSED_LOCKED AF_TRIGGER NOT_FOCUSED_LOCKED No effect
 NOT_FOCUSED_LOCKED AF_CANCEL INACTIVE Restart AF scan

AE and AWB state machines

The AE and AWB state machines are mostly identical. AE has additional FLASH_REQUIRED and PRECAPTURE states. So rows below that refer to those two states should be ignored for the AWB state machine.

mode = AE_MODE_OFF / AWB mode not AUTO
 State Transformation cause New state Notes
 INACTIVE AE/AWB disabled
 mode = AE_MODE_ON_* / AWB_MODE_AUTO
 State Transformation cause New state Notes
 INACTIVE HAL initiates AE/AWB scan SEARCHING
 INACTIVE AE/AWB_LOCK on LOCKED Values locked
 SEARCHING HAL finishes AE/AWB scan CONVERGED Good values, not changing
 SEARCHING HAL finishes AE scan FLASH_REQUIRED Converged but too dark without flash
 SEARCHING AE/AWB_LOCK on LOCKED Values locked
 CONVERGED HAL initiates AE/AWB scan SEARCHING Values locked
 CONVERGED AE/AWB_LOCK on LOCKED Values locked
 FLASH_REQUIRED HAL initiates AE/AWB scan SEARCHING Values locked
 FLASH_REQUIRED AE/AWB_LOCK on LOCKED Values locked
 LOCKED AE/AWB_LOCK off SEARCHING Values not good after unlock
 LOCKED AE/AWB_LOCK off CONVERGED Values good after unlock
 LOCKED AE_LOCK off FLASH_REQUIRED Exposure good, but too dark
 All AE states PRECAPTURE_START PRECAPTURE Start precapture sequence
 PRECAPTURE Sequence done, AE_LOCK off CONVERGED Ready for high-quality capture
 PRECAPTURE Sequence done, AE_LOCK on LOCKED Ready for high-quality capture
 Enable manual control
 Several controls are also involved in configuring the device 3A blocks to allow for direct app control.

The HAL model for 3A control is that for each request, the HAL inspects the state of the 3A control fields. If any 3A routine is enabled, then that routine overrides the control variables that relate to that routine, and these override values are then available in the result metadata for that capture. So for example, if autoexposure is enabled in a request, the HAL should overwrite the exposure, gain, and frame duration fields (and potentially the flash fields, depending on AE mode) of the request. The list of relevant controls is:

Control name	Unit	Notes
android.control.mode enum: OFF, AUTO, USE_SCENE_MODE		High-level 3A control. When set to OFF, all 3A control by the HAL is disabled. The app must set the fields for capture parameters itself. When set to AUTO, the individual algorithm controls in android.control.* are in effect,

such as android.control.afMode. When set to USE_SCENE_MODE, the individual controls in android.control.* are mostly disabled, and the HAL implements one of the scene mode settings (such as ACTION, SUNSET, or PARTY) as it wishes.

android.control.afMode enum OFF means manual control of lens focusing through android.lens.focusDistance.

android.control.aeMode enum OFF means manual control of exposure/gain/frame duration through android.sensor.exposureTime / .sensitivity / .frameDuration

android.control.awbMode enum OFF means manual control of white balance.

Thanks for sharing the full details — this is a rich overview of the **3A (Auto Focus, Auto Exposure, Auto White Balance)** systems and how their **modes and state transitions** are managed in the Android Camera HAL architecture.

Here's a breakdown of the key concepts you might be looking for, **especially around state transitions and configuration points**.

What Is "3A"?

3A refers to:

1. **AF (Auto Focus)** – Controls lens movement to get sharp images.
2. **AE (Auto Exposure)** – Adjusts brightness via ISO, exposure time, frame duration.
3. **AWB (Auto White Balance)** – Adjusts color temperature to make whites appear white.

Each of these has:

- A **Mode** (set by app/framework)
- A **State** (reported by HAL)
- A **Trigger** (to start actions)

3A State Machine: Where & When It Happens

When does 3A state machine activate?

- **After the camera is opened**, all 3A components are in the **INACTIVE** state.
- **3A doesn't reset** during stream reconfiguration (e.g., `configureStreams()`).
- 3A state changes **only when**:
 - App triggers a change (like **AF_TRIGGER_START**)
 - HAL automatically detects need to update AE/AWB/AF
 - User changes mode (e.g., from **AF_MODE_CONTINUOUS_VIDEO** to **AF_MODE_AUTO**)

Where Is 3A Controlled?

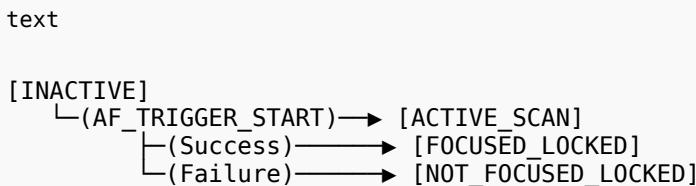
Layer	Role
App (Java/Kotlin)	Sets <code>android.control.*</code> keys in <code>CaptureRequest</code>
Framework	Passes down those keys to HAL
HAL	Implements the 3A logic + updates states in <code>CaptureResult</code>

Common Control Keys

Key	Function
<code>ANDROID_CONTROL_AF_MODE</code>	Set autofocus behavior (OFF, AUTO, CONTINUOUS_PICTURE, etc.)
<code>ANDROID_CONTROL_AF_TRIGGER</code>	Start/cancel AF
<code>ANDROID_CONTROL_AE_MODE</code>	Set exposure mode (ON, AUTO_FLASH, etc.)
<code>ANDROID_CONTROL_AE_LOCK</code>	Lock AE state
<code>ANDROID_CONTROL_AE_PRECAPTURE_TRIGGER</code>	Start AE metering before capture
<code>ANDROID_CONTROL_AWB_MODE</code>	Set white balance (AUTO, CLOUDY, etc.)
<code>ANDROID_CONTROL_AWB_LOCK</code>	Lock AWB values

AF Mode + State Transitions (Simplified)

Here's a quick example of how **AF state transitions** work in `AF_MODE_AUTO`:



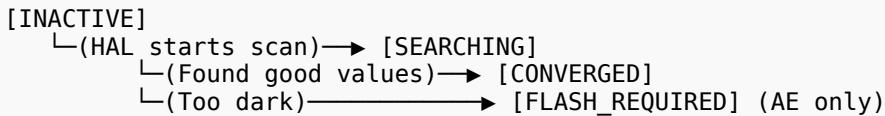
In `AF_MODE_CONTINUOUS_PICTURE`, HAL might *spontaneously* start scans and switch between:



AE/AWB State Transitions (Simplified)

They both follow a similar pattern:

text



They can be **locked** via:

- `ANDROID_CONTROL_AE_LOCK = true`
- `ANDROID_CONTROL_AWB_LOCK = true`

And unlocked to resume auto scanning.

Scene Modes & Overrides

- If `CONTROL_MODE == USE_SCENE_MODE`, then:
 - HAL decides the best combination of AE/AF/AWB modes.
 - App's selected AE/AF/AWB settings are ignored (except for `FACE_PRIORITY` mode).
-

Key Takeaways

- **3A state** lives **inside the HAL**, transitions driven by:
 - HAL logic
 - Triggers from the app
 - Mode switches
 - **App interacts via CaptureRequest keys**
 - **State is reported back in CaptureResult**
 - **Stream configuration does NOT reset 3A**
 - **Manual control is possible** by setting `CONTROL_MODE = OFF`
-

Would you like:

- A visual state diagram?
- A concrete code example in Java or C++?
- Explanation of how this is implemented in a custom HAL?

Let me know!



confusing, explain in simple terms

Sure! Let's simplify the Android **Camera 3A (AF, AE, AWB)** state machine and how it works.

What is "3A"?

When you take a photo or use the camera:

Term	What it does
AF (Auto Focus)	Makes sure the image is sharp (not blurry).
AE (Auto Exposure)	Makes the image not too dark or too bright.
AWB (Auto White Balance)	Makes colors look natural (e.g., white looks white).

Where is 3A used?

3A is used **before capturing an image** to make sure it's clear, bright, and has natural color. Android uses this when:

- You open the camera preview
- You focus or take a photo

Who controls 3A?

There are **3 players**:

1. **App (Java)** – Sends settings like “I want autofocus”
2. **Framework** – Passes them down
3. **Camera HAL (C++)** – Actually controls the lens, exposure, and color.

When does 3A start?

- After the app opens the camera and sets preview.
- It starts working **automatically** if you use `CONTROL_MODE_AUTO`.
- It runs **continuously** (if in continuous mode), or only when you **trigger it** (manual).

What are "states"?

Each 3A has a **state** — like a status.

Example:

AF can be:

- INACTIVE → Not doing anything
- SCANNING → Trying to focus
- FOCUSED → Got good focus
- FAILED → Could not focus

The **HAL** reports this back in `CaptureResult`.

Example: Auto Focus (AF)

Let's say you're using AF in manual mode (`AF_MODE_AUTO`):

1. Initially: **INACTIVE** (nothing is happening)
2. You tap to focus → app sends `AF_TRIGGER_START`
3. HAL starts focusing → **ACTIVE_SCAN**
4. If focus is good → **FOCUSED_LOCKED**
If not → **NOT_FOCUSED_LOCKED**

Now HAL is locked until you say `AF_TRIGGER_CANCEL`.

Key points

- App sets **modes** using keys like `android.control.afMode`
- HAL reads **these**, starts focusing, exposure, white balance
- HAL sends back **current state** in `CaptureResult`
- 3A **does not reset** when stream is reconfigured
- You can also lock AE or AWB using flags

Simple Summary

You do this...	3A does this...
Open camera	Starts working automatically
Tap to focus	AF starts scanning

You do this...	3A does this...
Take photo	AE + AWB make sure brightness and color are good
Lock exposure	Stops auto-adjusting light
Change mode	HAL resets only if mode changes

If you want, I can show a small flowchart or code snippet to make it clearer. Would that help?



Camera debugging

bookmark_border

This page describes the debugging tools in the camera service for viewing changes in capture request and result values that are sent to and from the camera HAL. The camera service includes the watch and the dumpsys commands. Available on devices running Android 13 or higher, the watch command allows control over when tags are monitored and accessed, live previewing of tag monitoring dumps from open clients, and viewing of cached dumps from closed clients. The dumpsys command allows for accessing debugging information but doesn't provide tag monitoring dumps from closed clients.

watch command

This section describes how to use the watch command and provides examples.

Start monitoring tags

To start monitoring tags, use:

```
adb shell cmd media.camera watch start -m <tags> [-c <clients>]
```

Example:

```
adb shell cmd media.camera watch start \
-m android.control.effectMode,android.control.aeMode \
-c com.google.android.GoogleCamera,com.android.chrome
```

Arguments:

tags: Comma-separated list of tags to be monitored. Also accepts the shorthand 3a, where 3a refers to the set of all android.control.* tags related to AF, AE, and AWB (for the full list of tags, see TagMonitor.cpp).

clients: Optional argument. Comma-separated list of client package names for which tags are monitored. Watches all clients if no client argument is passed, or if all is present in the list of clients.

This command starts tag monitoring in open clients and in any clients that are subsequently opened (until stop is called). After start is called, The camera service caches tag monitoring

dumps from clients when the clients close.

Unless start is called, the camera service doesn't monitor tags for any clients, and doesn't cache tag monitoring dumps. If all is passed in the clients parameter, the camera service monitors tags and caches monitoring dumps from all clients.

Dump tag monitoring information

To dump tag monitoring information, use:

```
adb shell cmd media.camera watch dump
```

This command dumps the tag monitoring information to standard output and then exits. It prints the cached tag monitoring dumps from clients that have closed since start (or last clear) and the latest tag monitoring dump from open clients.

Sample output:

```
$ adb shell cmd media.camera watch dump
Client: com.android.chrome (active)
1:com.android.chrome f0:532642803202286ns:      REQ:android.control.aeMode: [ON]
output stream ids: 0
1:com.android.chrome f0:532642803202286ns:      REQ:android.control.afMode:
[CONTINUOUS_PICTURE] output stream ids: 0
..
Client: com.google.android.GoogleCamera (cached)
0:com.google.android.GoogleCamera f0:532601698728552ns:
REQ:android.control.aeMode: [ON] output stream ids: 0 3 1 2
0:com.google.android.GoogleCamera f0:532601698728552ns:
REQ:android.control.afMode: [CONTINUOUS_PICTURE] output stream ids: 2 1 3 0
0:com.google.android.GoogleCamera f0:532601698728552ns:
REQ:android.control.awbMode: [AUTO] output stream ids: 0 3 1 2
...
Preview tag monitoring information in real time
```

To preview tag monitoring information in real time, use:

```
adb shell cmd media.camera watch live [-n refresh_interval_ms]
```

Example:

```
adb shell cmd media.camera watch live -n 250
```

Arguments:

refresh_interval_ms: Optional argument. Interval in milliseconds to refresh the information at. Defaults to 1000 if no value is passed.

This command prints tag monitoring information in real time. To exit, press return/enter.

Sample output:

```
$ adb shell cmd media.camera watch live
```

Press return to exit...

```
0:com.google.android.GoogleCamera f0:533016991302201ns:  
REQ:android.control.aeMode: [ON] output stream ids: 1 3 0 2  
0:com.google.android.GoogleCamera f0:533016991302201ns:  
REQ:android.control.afMode: [CONTINUOUS_PICTURE] output stream ids: 2 0 3 1  
...  
0:com.google.android.GoogleCamera f0:533017066793915ns:  
RES:android.control.aeState: [SEARCHING]  
0:com.google.android.GoogleCamera f0:533017066793915ns: 3  
RES:android.control.aeState: [SEARCHING]  
0:com.google.android.GoogleCamera f0:533017066793915ns: 2  
RES:android.control.aeState: [SEARCHING]  
0:com.google.android.GoogleCamera f0:533017066793915ns:  
RES:android.control.afState: [PASSIVE_SCAN]  
Clear cached dumps
```

To clear all cached tag monitoring dumps, use:

```
adb shell cmd media.camera watch clear
```

This command doesn't stop tag monitoring.

Stop monitoring tags

To stop monitoring tags in all clients and clear all buffers held for caching tag monitoring dumps, use:

```
adb shell cmd media.camera watch stop
```

dumps sys command

The dumps sys command provides a host of debugging information from the camera service.

The following command captures the entire debugging dump from the camera service:

```
adb shell dumps sys media.camera
```

The dumps sys command also allows capturing tag monitoring dumps from open clients.

However, dumps sys doesn't provide tag monitoring dumps from closed clients. The following are examples of using dumps sys for tag monitoring:

Capture tag monitoring dumps from all open clients:

```
adb shell dumpsys media.camera -m 3a | grep -A50 Monitored
Get live preview of tag monitoring information using the Linux watch command:
```

```
watch -n 1 -c 'adb shell dumpsys media.camera -m 3a | grep -A50 Monitored'
```

Great! Here's a **simple summary** of how to debug camera behavior using Android's **Camera Service** tools — especially helpful if you're working with Camera HAL, 3A (Auto Focus, Auto Exposure, Auto White Balance), or stream settings.

Why use camera debugging tools?

When your app or HAL code isn't behaving as expected — blurry images, exposure issues, or weird color tones — it's helpful to **see what control values are being used** and how the HAL responds.

Key Debug Tools

1. watch command (Android 13+)

Used to monitor and log **capture requests and results** in real-time.

What you can do:

- See **what your app requested** (e.g., AE = ON, AF = CONTINUOUS)
- See **what the camera HAL responded** with (e.g., AE state = SEARCHING)

Example: Start watching 3A values from specific apps

```
bash
adb shell cmd media.camera watch start -m 3a -c com.google.android.GoogleCamera
```

- **-m 3a**: monitor AE, AF, AWB-related tags
- **-c**: choose which app(s) to monitor

Dump monitored data (from open & closed clients)

```
bash
adb shell cmd media.camera watch dump
```

● View live 3A values (real-time output, auto-refresh)

```
bash
adb shell cmd media.camera watch live -n 250
```

Refreshes every 250 ms — helpful during preview or photo capture.

✓ Clear cached monitoring data (but keeps watching)

```
bash
adb shell cmd media.camera watch clear
```

ⓧ Stop watching everything

```
bash
adb shell cmd media.camera watch stop
```

2. dumpsys command

Works on **all Android versions**, but gives **less detailed tag monitoring** than `watch`.

Full camera debug info:

```
bash
adb shell dumpsys media.camera
```

Show 3A tag monitoring (for open clients only):

```
bash
adb shell dumpsys media.camera -m 3a | grep -A50 Monitored
```

Watch it update every second (Linux/macOS only):

```
bash
watch -n 1 -c 'adb shell dumpsys media.camera -m 3a | grep -A50 Monitored'
```

In Plain English

- Use `watch` to see exactly **what the app tells the camera** (request) and **what the HAL does in return** (result).
- Useful for debugging things like:
 - Why the camera won't focus
 - Why exposure doesn't stabilize
 - Why images are too dark/light
- `watch` keeps logs **even after the app closes**, but `dumppsys` does not.

Would you like a quick visual diagram of this process or a cheat sheet?



Is 3A work simultaneously

Yes — **3A (Auto Exposure, Auto Focus, and Auto White Balance) work simultaneously** and in coordination. Here's how, explained simply:

What is 3A?

Component	Purpose	Example
AE	Auto Exposure	Controls brightness
AF	Auto Focus	Makes subject sharp
AWB	Auto White Balance	Corrects color tint (e.g., yellowish)

Do they run at the same time?

 Yes — during **preview and capture**, Android's camera framework requests all 3 controls in parallel. The **Camera HAL** and underlying **ISP process all 3A algorithms together per frame**.

How 3A work together per frame:

1. App sends a capture request:

- "Give me a preview frame with AE=ON, AF=CONTINUOUS, AWB=ON"

2. Camera HAL processes the request:

- Exposure, focus, and white balance algorithms analyze the sensor data from that frame.

3. Each algorithm updates its metadata:

- AE might adjust shutter speed
- AF might move lens position
- AWB might shift color gains

4. Results are reported back in metadata

And this happens **continuously** — for every frame during preview or video.

Are they dependent?

- AE and AWB usually **settle quickly**, within a few frames.
- AF might **take longer**, especially in low light.
- In some cases, AE or AWB may **wait for AF to complete** (like in HDR or night mode).

You can observe this live with:

```
bash  
  
adb shell cmd media.camera watch live -m 3a
```

This shows you:

- AE mode and AE state (**SEARCHING, CONVERGED**)
- AF mode and AF state (**ACTIVE_SCAN, FOCUSED**)
- AWB mode and state (**CONVERGED**, etc.)

Let me know if you want a visual timeline showing how all 3A components evolve frame by frame.



at what frequency they run

Great question!

At what frequency do 3A (AE, AF, AWB) run?

They run **once per frame**, i.e., **at the camera's frame rate**.

So if the camera is running at 30 FPS:

- AE, AF, and AWB all **analyze and adjust every ~33 milliseconds** (1 / 30).
- This ensures the 3A system keeps up with rapid scene changes.

More detail:

3A Component	Runs Per Frame?	Can Skip Frames?	Adaptive Frequency?
AE	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> Usually not	Can adapt in burst modes
AF	<input checked="" type="checkbox"/> Yes (if active)	<input checked="" type="checkbox"/> Sometimes	Continuous AF may slow down
AWB	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> Usually not	Rarely adaptive

Summary:

- 3A runs **per frame** in the **camera pipeline**.
- The **frequency equals the preview or video stream frame rate** — typically 30, 60, or even 120 FPS.
- All 3 work in **parallel** within each frame processing loop.

If you want, I can show you a **per-frame timeline** with AE/AF/AWB decisions changing over time. Would that help?



how their values calculated and used

Excellent — let's break it down simply:

How 3A Values Are Calculated and Used (AE, AF, AWB)

Each **3A algorithm** runs **inside the camera HAL or ISP** and uses **image statistics** from the sensor. Here's how each works:

1. AE – Auto Exposure

What it does:

AE decides:

- Exposure time (shutter speed)
- ISO gain (sensor sensitivity)

- Sometimes frame duration (FPS limit)

How AE calculates values:

1. Sensor sends raw image.
2. ISP generates **histogram / luminance map**.
3. AE algorithm checks if image is too bright or dark.
4. Adjusts exposure/gain to bring average brightness into a **target range** (e.g., 18% gray).

How it's used:

- AE updates are sent in `CaptureRequest` metadata (`android.sensor.exposureTime`, `android.sensor.sensitivity`).
- These values **take effect in future frames** (1–2 frame delay).
- App can lock AE (e.g., before taking a photo).

2. AF – Auto Focus

What it does:

Moves the lens to make the subject sharp.

How AF calculates focus:

1. ISP runs **contrast-detection** (or PDAF phase detection).
2. For contrast AF:
 - Lens steps through positions.
 - At each step, calculates **sharpness metric** (edge strength).
 - Finds peak sharpness = best focus.
3. For PDAF:
 - Uses special pixels to estimate depth quickly.

How it's used:

- Lens position is set via `android.lens.focusDistance`.
- App sees focus state via `CaptureResult` (`FOCUSED`, `NOT_FOCUSED`, etc).

3. AWB – Auto White Balance

What it does:

Corrects color cast from lighting (e.g., daylight, tungsten, fluorescent).

How AWB calculates values:

1. ISP computes **RGB histograms** or average color per region.
2. AWB estimates **color temperature** of the light (e.g., 6500K for daylight).
3. It adjusts **color gains** to make neutral objects (e.g., gray) truly neutral.

How it's used:

- Sets `android.colorCorrection.gains` and `transform`.
- Affects how the RAW is demosaiced or YUV is generated.
- Can be locked (e.g., for consistent video).

When are values applied?

Frame N	3A algorithms run (on frame N-1 stats)
Frame N+1	Updated values are applied in request

 There's always a **1-2 frame delay** between stats → calculation → application.

App-Level Interaction

- App **requests 3A** via `CaptureRequest`:

```
java
captureRequestBuilder.set(CaptureRequest.CONTROL_AE_MODE, AE_MODE_ON);
```

- App **gets 3A results** from `CaptureResult`:

```
java
captureResult.get(CaptureResult.CONTROL_AF_STATE);
```

Would you like to see a real example of `CaptureResult` metadata showing these values?



yes

Great! Here's a **realistic example** of how 3A values appear in a `CaptureResult` for a frame captured by the Android camera HAL:

Example: CaptureResult Metadata

java

```
CaptureResult result = captureCallbackResult;

Long exposureTime = result.get(CaptureResult.SENSOR_EXPOSURE_TIME); // in nanoseconds
Integer iso = result.get(CaptureResult.SENSOR_SENSITIVITY); // ISO gain
Float focusDistance = result.get(CaptureResult.LENS_FOCUS_DISTANCE); // diopters (1/m)

Integer aeState = result.get(CaptureResult.CONTROL_AE_STATE); // AE state enum
Integer afState = result.get(CaptureResult.CONTROL_AF_STATE); // AF state enum
Integer awbState = result.get(CaptureResult.CONTROL_AWB_STATE); // AWB state enum

float[] whiteBalanceGains = result.get(CaptureResult.COLOR_CORRECTION_GAINS).getArray();
```

What these values might look like in real output:

Field	Example Value	Meaning
<code>SENSOR_EXPOSURE_TIME</code>	<code>10000000</code> ns (10ms)	1/100s shutter
<code>SENSOR_SENSITIVITY</code>	<code>200</code>	ISO 200
<code>LENS_FOCUS_DISTANCE</code>	<code>0.5f</code>	Focus at 2 meters (1 / 0.5)
<code>CONTROL_AE_STATE</code>	<code>CONVERGED</code>	AE locked onto target brightness
<code>CONTROL_AF_STATE</code>	<code>FOCUSED_LOCKED</code>	Focus completed
<code>CONTROL_AWB_STATE</code>	<code>CONVERGED</code>	White balance is correct
<code>COLOR_CORRECTION_GAINS</code>	<code>[2.1, 1.0, 1.0, 1.8]</code>	RGGB color gains for white balance

How they change over time:

- These values can **change every frame** if 3A is active.
- You can observe them in a `CameraCaptureSession.CaptureCallback`:

java

```
@Override
public void onCaptureCompleted(CameraCaptureSession session,
                               CaptureRequest request,
                               TotalCaptureResult result) {
    Log.d("AE", "ExposureTime: " + result.get(CaptureResult.SENSOR_EXPOSURE_TIME));
```

```

Log.d("AF", "FocusState: " + result.get(CaptureResult.CONTROL_AF_STATE));
Log.d("AWB", "Gains: " +
Arrays.toString(result.get(CaptureResult.COLOR_CORRECTION_GAINS).getArray()));
}

```

Would you like me to simulate a series of frames showing how 3A evolves during auto adjustment (e.g., as brightness changes or focus locks)?



Common Control Keys

Key Function

ANDROID_CONTROL_AF_MODE	Set autofocus behavior (OFF, AUTO, CONTINUOUS_PICTURE, etc.)
ANDROID_CONTROL_AF_TRIGGER	Start/cancel AF
ANDROID_CONTROL_AE_MODE	Set exposure mode (ON, AUTO_FLASH, etc.)
ANDROID_CONTROL_AE_LOCK	Lock AE state
ANDROID_CONTROL_AE_PRECAPTURE_TRIGGER	Start AE metering before capture
ANDROID_CONTROL_AWB_MODE	Set white balance (AUTO, CLOUDY, etc.)
ANDROID_CONTROL_AWB_LOCK	Lock AWB values

Excellent — that's a good summary of **commonly used Camera2 API controls**. Let me explain each one in **very simple terms** and how they are used practically in your camera app:

Common 3A Control Keys Explained

Key	Simple Meaning	Typical Use Case
ANDROID_CONTROL_AF_MODE	Sets how autofocus behaves (e.g., manual, continuous, or on-demand).	Use CONTINUOUS_PICTURE for live preview or photos.
ANDROID_CONTROL_AF_TRIGGER	Tells the camera to start or stop autofocus manually.	Use with AF_MODE=AUTO to trigger a single autofocus pass.
ANDROID_CONTROL_AE_MODE	Sets how auto-exposure works (e.g., with/without flash).	ON = auto exposure without flash.
ANDROID_CONTROL_AE_LOCK	Locks current exposure (brightness) settings.	Use this after exposure has stabilized to avoid changes.
ANDROID_CONTROL_AE_PRECAPTURE_TRIGGER	Starts a short brightening to help AE settle before capturing.	Required before high-quality still image capture.
ANDROID_CONTROL_AWB_MODE	Sets how auto white balance works (e.g., AUTO, DAYLIGHT, CLOUDY).	AUTO is best for general use; manual if lighting is fixed.

Key	Simple Meaning	Typical Use Case
ANDROID_CONTROL_AWB_LOCK	Locks current white balance setting.	Avoids color shift mid-capture.

🧠 Quick Examples

💡 Autofocus (AF):

java

```
// Set AF mode to continuous
builder.set(CaptureRequest.CONTROL_AF_MODE, CaptureRequest.CONTROL_AF_MODE_CONTINUOUS_PICTURE);
```

📸 Trigger one-time focus:

java

```
// Set AF mode to AUTO and trigger focus
builder.set(CaptureRequest.CONTROL_AF_MODE, CaptureRequest.CONTROL_AF_MODE_AUTO);
builder.set(CaptureRequest.CONTROL_AF_TRIGGER, CaptureRequest.CONTROL_AF_TRIGGER_START);
```

🔒 Lock AE/AWB:

java

```
builder.set(CaptureRequest.CONTROL_AE_LOCK, true);
builder.set(CaptureRequest.CONTROL_AWB_LOCK, true);
```

Would you like a full **real capture request setup** showing these controls in use together?



They use ISP to execute their algorithm

Yes, you're absolutely right — **3A (Auto Focus, Auto Exposure, Auto White Balance)** typically rely on the **ISP (Image Signal Processor)** to run their algorithms efficiently and in real time.

✓ Here's how the 3A algorithms work with the ISP:

3A Component	What it does	What ISP does for it
AF (Autofocus)	Finds the best lens position to make the image sharp	ISP analyzes contrast or phase data from the image to compute focus quality.

3A Component	What it does	What ISP does for it
AE (Auto Exposure)	Adjusts brightness (ISO, shutter, aperture)	ISP calculates image brightness from sensor data, applies histogram or region-based AE.
AWB (Auto White Balance)	Corrects color tones to look natural (e.g., removing blue from indoor light)	ISP analyzes image color channels to compute gains and adjust them in hardware.

⚙️ Why ISP is used:

- **Speed:** ISPs are hardware-accelerated and optimized for real-time processing (especially for video).
- **Parallelism:** ISP can analyze image statistics for AF, AE, and AWB simultaneously.
- **Power Efficiency:** Hardware execution saves power compared to doing the same in CPU or GPU.
- **Image Metadata:** ISP provides **statistics** (like brightness histograms, focus scores, color balance) used in next-frame decision making.

📦 ISP Outputs Used by 3A:

- **AF stats:** Sharpness/contrast per region (or phase-difference if PDAF)
- **AE stats:** Luminance histogram, average brightness
- **AWB stats:** Per-region R/G/B intensity data

These stats are captured in the background by ISP during **preview or capture**, and then fed into the 3A algorithm to compute the **next control values**, which are sent back as part of a future **CaptureRequest**.

Would you like to see a **flow diagram** or a **step-by-step of how 3A uses ISP output and updates requests?**



So they do not use CPU?

Great question. Here's a simple and clear breakdown:

🔄 Do 3A algorithms use the CPU?

✓ Yes, but **not for everything**.

🧠 What uses the CPU:

- **The actual 3A algorithm logic** (e.g., computing best exposure, deciding new lens position, calculating white balance gains) runs on the **CPU** or **DSP** (sometimes).
- These are typically lightweight, frame-by-frame calculations.
- The **Camera HAL** runs this logic, possibly using CPU threads or dedicated camera firmware.

What uses the ISP:

- The **image statistics** (sharpness, brightness, color data) are collected by the **ISP hardware**.
- ISP gives fast, parallel data from the image sensor without involving CPU much.
- ISP can also apply hardware changes (e.g., lens shading correction, white balance gains) to the output frames.

Flow Summary:

1. Sensor → ISP

ISP processes raw image → extracts statistics (AF, AE, AWB)

2. ISP → Camera HAL (via CPU)

Camera HAL receives stats via CPU → runs 3A algorithm in software

3. Camera HAL → Next CaptureRequest

HAL sets new values: lens position (AF), exposure time (AE), color gains (AWB)

4. Next frame → ISP

ISP uses new settings for actual image processing

Who does what?

Component	Role
ISP	Fast, hardware image stats collection & image processing
CPU (via Camera HAL)	Runs 3A logic based on ISP stats, sets control parameters
GPU	Not involved in 3A (used for display or image effects only)

So, 3A **does need the CPU**, but it **depends heavily on ISP** for fast stats and hardware support.

Let me know if you want a **diagram** of this interaction!



Error and stream handling

bookmark_border

Error management

HIDL interface methods that interact with the camera must generate the corresponding camera-specific status.

If `ICameraDeviceCallbacks::notify()` is called with `ERROR_DEVICE`, only the `ICameraDeviceSession::close()` method can be called successfully. All other methods will return `INTERNAL_ERROR`.

Transient errors in image capture must be reported through `ICameraDeviceCallbacks::notify()` with the appropriate error code. In each transient failure case, the HAL must still call `ICameraDeviceCallbacks::processCaptureResult()` with an appropriate capture result.

Stream management

HAL clients must configure camera streams by calling `ICameraDeviceSession::configurestreams()`.

Let's break this down in simple terms by separating **Error Handling** and **Stream Handling** in the Android Camera HAL (HIDL) layer:

1. Error Handling in Camera HAL (HIDL)

What is the purpose?

To **tell the framework** (e.g., CameraService / app) that something **went wrong**, either *temporarily* or *permanently*.

Two types of errors:

a. Fatal Error (Device-level)

- **Reported via:** `ICameraDeviceCallbacks::notify(ERROR_DEVICE)`
- **What happens next:**
 - All camera functions **stop** working.
 - Only `ICameraDeviceSession::close()` will work.
 - Any other method call will return `INTERNAL_ERROR`.
-  Example: Camera hardware crashes, HAL internal failure.

b. Transient Error (Temporary issues)

- **Reported via:** `ICameraDeviceCallbacks::notify()` with codes like:

- **ERROR_REQUEST**
 - **ERROR_RESULT**
 - **ERROR_BUFFER**
 - These indicate that a **specific request or buffer failed**, but the camera can **continue working**.
 - In these cases, HAL **must still call processCaptureResult()** — even if partial — to avoid leaving the pipeline hanging.
-

2. Stream Handling (configureStreams)

What is a stream?

A **pipeline** through which image data flows:

- Preview stream (YUV or RGB)
 - JPEG capture stream
 - Depth stream
 - Video stream
-

ICameraDeviceSession::configureStreams():

This is called by the framework to tell HAL:

“I want to use these streams. Please set them up.”

When is configureStreams() used?

- When a camera is **opened** for the first time
 - When an app changes the **resolution**, format, or stream type
 - When reconfiguring after session failure
-

HAL responsibilities:

- Allocate buffers or register with gralloc
- Inform ISP or image pipeline about each stream's:
 - Resolution
 - Format (YUV, JPEG, RAW, etc.)
 - Usage flags (GPU, video encoder, etc.)

Once configured:

- HAL receives `CaptureRequests` that use these streams.
- HAL fills and returns output buffers through `processCaptureResult()`.

Summary

Aspect	Key API	Meaning
Fatal error	<code>notify(ERROR_DEVICE)</code>	Stop camera session. Only <code>close()</code> works.
Transient error	<code>notify(ERROR_*) + processCaptureResult()</code>	Inform framework of temporary issues but continue.
Stream setup	<code>configureStreams()</code>	HAL sets up image pipelines, formats, buffers.
Stream usage	<code>processCaptureRequest() → processCaptureResult()</code>	Frames flow through active streams.

Let me know if you want a **diagram** or timeline of error + stream handling interactions!



Metadata and controls

`bookmark_border`

Metadata support

To support the saving of raw image files by the Android framework, substantial metadata is required about the sensor's characteristics. This includes information such as color spaces and lens shading functions.

Most of this information is a static property of the camera subsystem and can therefore be queried before configuring any output pipelines or submitting any requests. The new camera APIs greatly expand the information provided by the `getCameraInfo()` method to provide this information to the app.

In addition, manual control of the camera subsystem requires feedback from the assorted devices about their current state, and the actual parameters used in capturing a given frame. The actual values of the controls (exposure time, frame duration, and sensitivity) as actually used by the hardware must be included in the output metadata. This is essential so that apps know when either clamping or rounding took place, and so that the app can compensate for the real settings used for image capture.

For example, if an app sets frame duration to 0 in a request, the HAL must clamp the frame duration to the real minimum frame duration for that request, and report that clamped

minimum duration in the output result metadata.

So if an app needs to implement a custom 3A routine (for example, to properly meter for an HDR burst), it needs to know the settings used to capture the latest set of results it has received to update the settings for the next request. Therefore, the new camera API adds a substantial amount of dynamic metadata to each captured frame. This includes the requested and actual parameters used for the capture, as well as additional per-frame metadata such as timestamps and statistics generator output.

Per-setting control

For most settings, the expectation is that they can be changed every frame, without introducing significant stutter or delay to the output frame stream. Ideally, the output frame rate should solely be controlled by the capture request's frame duration field, and be independent of any changes to processing blocks' configuration. In reality, some specific controls are known to be slow to change; these include the output resolution and output format of the camera pipeline, as well as controls that affect physical devices, such as lens focus distance. The exact requirements for each control set are detailed later.

Raw sensor data support

In addition to the pixel formats supported by the old API, the new API adds a requirement for support for raw sensor data (Bayer RAW), both for advanced camera apps as well as to support raw image files.

Let's break this down into **simple terms**:

What is Metadata in Camera?

Metadata is extra information that comes **with each image** or **from the camera**.

It tells you:

- How the image was taken
- What settings were used
- What the hardware is capable of

Types of Metadata

1. Static Metadata (comes before you start capturing)

Describes the **capabilities** of the camera:

- Supported resolutions
- Available exposure range

- Lens info (field of view, distortion, etc.)
- Supported formats (JPEG, RAW, etc.)

→ Queried via: `getCameraInfo()` or `CameraCharacteristics`

👉 This **doesn't change** per frame.

2. Dynamic Metadata (comes with each captured frame)

Describes:

- What settings were **requested** by the app
- What settings were **actually used** by hardware (in case of clamping/rounding)
- Frame timestamps
- 3A statistics (AF, AE, AWB feedback)

→ Comes via: `processCaptureResult()`

👉 This **changes every frame**.

⌚ Why This Is Important?

If your app wants to control the camera **manually** (especially for things like HDR, long exposure, etc.), you **must know exactly** what was done in the previous frame.

You'll use that feedback to decide how to change settings for the **next frame**.



Per-setting Controls

Most settings can be changed **for every frame**. For example:

- Exposure time
- ISO
- White balance
- Focus trigger

But **some settings are slow to change**, such as:

- Lens physical movement (autofocus)
- Stream resolution or format changes
(These require reconfiguring streams, causing delays.)

RAW Sensor Support

- The new camera APIs support capturing **RAW Bayer images**.
- This is useful for pro apps and photography apps that want to process RAW images.

Summary Table

Metadata Type	When Available	Example Info	Why Needed
Static	Before configuration	Lens info, formats, max ISO	Let app know what camera can do
Dynamic	Per frame	Exposure used, focus state, 3A stats	Feedback to tune next request
RAW Data	During capture	RAW Bayer image	For full image processing in apps

Let me know if you want diagrams or examples for each part.



Output streams, cropping, and zoom

bookmark_border

Output streams

The camera subsystem operates solely on the ANativeWindow-based pipeline for all resolutions and output formats. Multiple streams can be configured at one time to send a single frame to many targets such as the GPU, the video encoder, RenderScript, or app-visible buffers (RAW Bayer, processed YUV buffers, or JPEG-encoded buffers).

As an optimization, these output streams must be configured ahead of time, and only a limited number may exist at once. This allows for pre-allocation of memory buffers and configuration of the camera hardware, so that when requests are submitted with multiple or varying output pipelines listed, there won't be delays or latency in fulfilling the request.

For further information about the guaranteed stream output combinations that depend on the supported hardware level, see `createCaptureSession()`.

Cropping

Cropping of the full pixel array (for digital zoom and other use cases where a smaller FOV is desirable) is communicated through the `ANDROID_SCALER_CROP_REGION` setting. This is a per-request setting, and can change on a per-request basis, which is critical for implementing smooth digital zoom.

The region is defined as a rectangle ($x, y, \text{width}, \text{height}$), with (x, y) describing the top-left corner of the rectangle. The rectangle is defined on the coordinate system of the sensor active pixel array, with $(0,0)$ being the top-left pixel of the active pixel array. Therefore, the width and height cannot be larger than the dimensions reported in the `ANDROID_SENSOR_ACTIVE_PIXEL_ARRAY` static info field. The minimum allowed width and height are reported by the HAL through the `ANDROID_SCALER_MAX_DIGITAL_ZOOM` static info field, which describes the maximum supported zoom factor. Therefore, the minimum crop region width and height are:

```
{width, height} =  
{ floor(ANDROID_SENSOR_ACTIVE_PIXEL_ARRAY[0] /  
    ANDROID_SCALER_MAX_DIGITAL_ZOOM),  
  floor(ANDROID_SENSOR_ACTIVE_PIXEL_ARRAY[1] /  
    ANDROID_SCALER_MAX_DIGITAL_ZOOM) }
```

If the crop region needs to fulfill specific requirements (for example, it needs to start on even coordinates, and its width/height needs to be even), the HAL must do the necessary rounding and write out the final crop region used in the output result metadata. Similarly, if the HAL implements video stabilization, it must adjust the result crop region to describe the region actually included in the output after video stabilization is applied. In general, a camera-using app must be able to determine the field of view it is receiving based on the crop region, the dimensions of the image sensor, and the lens focal length.

Since the crop region applies to all streams, which may have different aspect ratios than the crop region, the exact sensor region used for each stream may be smaller than the crop region. Specifically, each stream should maintain square pixels and its aspect ratio by minimally further cropping the defined crop region. If the stream's aspect ratio is wider than the crop region, the stream should be further cropped vertically, and if the stream's aspect ratio is narrower than the crop region, the stream should be further cropped horizontally.

In all cases, the stream crop must be centered within the full crop region, and each stream is only either cropped horizontally or vertical relative to the full crop region, never both.

For example, if two streams are defined, a 640x480 stream (4:3 aspect), and a 1280x720 stream (16:9 aspect), below demonstrates the expected output regions for each stream for a few sample crop regions, on a hypothetical 3 MP (2000 x 1500 pixel array) sensor.

Crop region: (500, 375, 1000, 750) (4:3 aspect ratio)
 640x480 stream crop: (500, 375, 1000, 750) (equal to crop region)
 1280x720 stream crop: (500, 469, 1000, 562)

Yes, this section explains **how the Android Camera HAL handles output streams, digital zoom, and cropping** for multi-stream capture pipelines.

Let's break it down:

✓ 1. Output Streams (multi-target rendering)

- The camera HAL supports **multiple output streams** (think: JPEG, YUV, RAW, etc.).
- These are all based on `ANativeWindow` surfaces (e.g., `SurfaceView`, `SurfaceTexture`).
- **Each stream can go to different components**, such as:
 - GPU (for rendering)
 - Video encoder (for recording)
 - App buffers (for processing in RenderScript or OpenGL)
- This enables **one frame** to be sent to **multiple consumers simultaneously**.

🔧 Pre-configuration Required:

- All output streams must be configured **in advance** using `configureStreams()`.
- This avoids **latency** by allowing the HAL to pre-allocate buffers and hardware configuration.

✓ 2. Cropping for Digital Zoom

- Cropping is used to **zoom digitally** or reduce field-of-view (FoV).
- This is done using the `ANDROID_SCALER_CROP_REGION` metadata field.
 - Format: `(x, y, width, height)`
 - Origin `(0, 0)` is top-left of the full sensor active array.

📏 Crop bounds:

- Max region: full sensor area (`ANDROID_SENSOR_ACTIVE_PIXEL_ARRAY`)
- Min region: depends on `ANDROID_SCALER_MAX_DIGITAL_ZOOM`
 - $\text{Min width} = \text{sensor_width} / \text{zoom}$
 - $\text{Min height} = \text{sensor_height} / \text{zoom}$

💡 Example (3MP sensor: 2000x1500, max zoom: 2x):

- Min crop region = 1000×750

✓ 3. Stream-specific Cropping

Each output stream:

- Inherits the **overall crop region**

- May **apply further cropping** to maintain its aspect ratio (e.g., 16:9, 4:3)
- Maintains **square pixels**

Rules:

- Aspect mismatch leads to **further crop** (horizontal or vertical only)
- Stream crop must be **centered** in the global crop
- Never both cropped horizontally *and* vertically simultaneously

Example:

Let's say:

- Sensor = 2000x1500 (4:3)
- Crop region = (500, 375, 1000, 750) → 4:3
- Streams:
 - 640x480 (4:3): no extra crop needed
 - 1280x720 (16:9): HAL adjusts vertical crop to maintain 16:9, becomes:
 - (500, 469, 1000, 562) → crop top/bottom within full crop

Why this matters:

- Ensures consistent **FoV** and **zoom control**
- Maintains high performance with **zero runtime stream reconfiguration**
- Apps can deduce final crop from metadata for overlays, zoom UX, etc.

Let me know if you want a **visual diagram** or a breakdown of how HAL should report the adjusted crop in **CaptureResult**.





This image illustrates how digital zoom and multiple output streams work in the Android Camera HAL using a crop-based approach. Let me break it down simply:

Key Concepts Shown in the Diagram

1. Active Pixel Array

- The **full sensor resolution** (e.g., 2000x1500 in this example).
- This is the maximum area from which the camera can read raw data.

2. Crop Region (Blue Box - 2000x1500)

- Represents the **actual region** being read from the sensor for a given frame.
- The crop region size can change based on **zoom** or field-of-view (FoV) changes.

3. Zoom Field-of-View (Purple Box - 2.0x Zoom)

- When you zoom in (e.g., 2x), the camera doesn't physically zoom — instead it **shrinks the crop region**, selecting a smaller area of the sensor.
- A 2.0x zoom means you're selecting a **central quarter of the sensor area**, discarding outer areas, and upscaling digitally.

4. Output Streams (Green Boxes)

- **640x480 stream** and **1280x720 stream** are two separate camera output resolutions.
- The HAL must map the selected crop region to each output stream size.
- Output resolutions are **scaled** from the cropped area depending on stream configuration.

How This Works Together

- The app or framework sets a **zoom ratio** (like 2.0x).
- HAL calculates the **crop region** based on that zoom.
- The sensor captures only the region inside that crop box.
- HAL then **scales** that region into the requested output resolutions (e.g., 720p and VGA).
- These streams can be used simultaneously for preview, video recording, or analysis.

Why This Matters

- Enables **real-time zoom** without mechanical lenses.
- Supports **multiple resolutions** simultaneously (multi-stream support).
- Efficient: reduces bandwidth by not reading unnecessary sensor data.

If you'd like, I can explain how this is configured via Camera HAL metadata (`SCALER_CROP_REGION`, `ANDROID_SCALER_AVAILABLE_STREAM_CONFIGURATIONS`, etc.).



confusing

No problem — let's simplify this image **step by step**:

What Is This a Picture Of?

It's a **camera sensor** diagram showing:

- Zoom
- Cropping
- Stream output sizes

1. What is the "Active Pixel Array"?

This is the **entire camera sensor area** (the big gray background).

The camera can see this whole area, but it doesn't always use the full thing.

2. What is the Blue Box (2000x1500)?

This is the **crop region**:

The camera chooses **only part of the sensor** to use when taking a picture or video.

Think of it as the area the camera is "looking at."

3. What is the Purple Box (2.0 zoom)?

This shows what happens when you **zoom in 2x**:

- The camera **shrinks the visible area** (purple box is smaller than sensor).
- It picks a **smaller rectangle in the center**.
- This is called **digital zoom** (not optical).

 You're not moving a lens — you're just using a smaller part of the image and enlarging it.

4. What are the Green Boxes (640x480 and 1280x720)?

These are the **output image sizes**.

- The camera takes the cropped image (inside purple box),
- Then **resizes it** to different resolutions:
 - One stream is 640x480 (VGA),
 - Another is 1280x720 (HD).

These are what apps like Camera or Zoom actually receive.

CAMERA Why This Happens

You may ask:

Why zoom like this? Why not just take the full image?

Because:

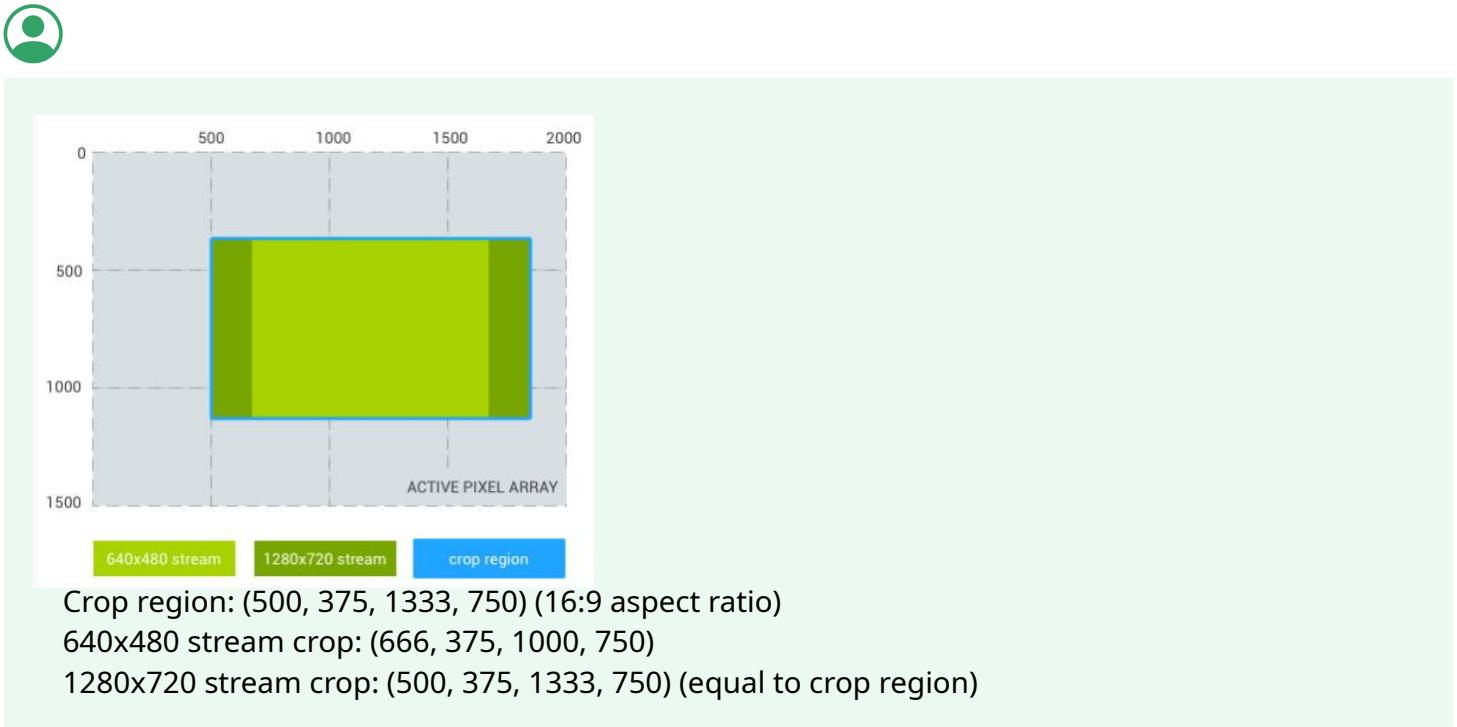
- Zoom makes you see **closer** by cropping.
- Output streams need **smaller sizes** (HD, VGA), not the full sensor.

✓ Summary

Concept	Box Color	Meaning
Sensor area	Gray	Full camera sensor (active pixel array)

Concept	Box Color	Meaning
Crop region	Blue	Area chosen for capture
2x zoom	Purple	Smaller area → digital zoom in
Output streams	Green	Final image sizes the app gets

Would you like this explained with a simple camera zoom analogy (like phone camera zoom)?



Great! Let's explain this updated image clearly.

🔍 What This Shows

This is a **visual layout** of:

- A full camera **sensor area** (gray)
- A **crop region** (blue) with **16:9 aspect ratio**
- Two **output streams**:
 - **1280x720** (light green)
 - **640x480** (dark green)

📘 Crop Region (Blue Box)

- 🔎 Coordinates: (500, 375, 1333, 750)

- This defines a **rectangle inside the sensor** where image data is read.
- It's a **16:9 crop** of the sensor (like widescreen video).

This is the base region used for capturing the image.

1280x720 Stream (Light Green)

- Crop: **Same as the crop region**: (500, 375, 1333, 750)
- Aspect Ratio: **16:9**
- This means **no additional crop** is applied — it directly uses the selected region and **scales it down** to 1280×720 resolution.

640x480 Stream (Dark Green)

- Crop: (666, 375, 1000, 750)
- Aspect Ratio: **4:3**
- This stream:
 - **Narrower** than the main crop region (cutting sides).
 - Keeps same height (375–750) but **cropped in horizontally**.
 - This allows the stream to match 4:3 shape even though the original crop was 16:9.

Summary Table

Stream	Crop Coords	Aspect Ratio	Notes
Crop region	(500, 375, 1333, 750)	16:9	Base input region (used by default)
1280x720	(500, 375, 1333, 750)	16:9	Full crop region, scaled down
640x480	(666, 375, 1000, 750)	4:3	Narrower crop, fits 4:3 output

Why Crop Like This?

Each stream must **match its aspect ratio**, so:

- The 1280x720 stream can use the full crop.
- The 640x480 stream **must crop horizontally** to avoid stretching/squashing.

Would you like to see how this would work during **zooming or preview + capture flow?**



Why do we have crop region? explain in simple words

Great question! Let's break it down **super simple:**

What is a "Crop Region" in Camera?

 Think of your camera sensor like a big photo:

- It's bigger than what you always need.
- Sometimes you only want to use a **part of that photo**.

That part is called the **crop region**.

Why Use a Crop Region?

1. To Zoom

- When you "zoom in," you're not moving the lens.
- You just select a **smaller part** of the sensor (crop) and scale it up.
- That's called **digital zoom**.

 Smaller crop → zoomed-in view

 Bigger crop → zoomed-out view

2. To Match Aspect Ratios

- You might need one stream in **16:9** (widescreen) and another in **4:3** (photo).
- The HAL sets different crop regions to get the right shape from the sensor.

3. To Improve Quality

- By cropping only the center, you can avoid blurry or noisy edges of the sensor.

Example

If the full sensor is this big:

css

[FULL SENSOR IMAGE]

But you only want to capture this middle part:

css

[CROP REGION]

Then the camera HAL tells the hardware:

"Hey! Just read this **crop region** from the sensor!"

✓ Summary

Purpose	How Crop Region Helps
Zoom	Selects smaller area = zoomed image
Aspect Ratio	Picks region to match screen shape
Quality	Focuses on the best part of sensor
Performance	Reads less data = faster processing

Let me know if you want to see it with a simple diagram!



Why do we have multiple crop streams?

Excellent question — let's keep it **simple** and build on the previous explanation.

🤔 Why do we have multiple crop streams?

Because the camera often needs to send **different versions** of the same image to **different places, at the same time**.



Real-world example:

You take a photo using a camera app — here's what happens:

Use Case	Image Size	Why It's Needed
Preview on screen	640x480 (small)	Fast and smooth UI display (low latency)
Recording Video	1280x720 (HD)	Good quality, real-time video
Capture Photo	4000x3000 (large)	High-quality image for saving

So why different crop regions?

Each stream might:

- **Want a different size** (e.g., 640x480 vs. 1280x720)
- **Have a different shape** (aspect ratio: 4:3, 16:9)
- **Zoom differently** (more zoom = tighter crop)

Since all these streams use the **same sensor**, Android's camera HAL needs to **define a crop region** for each stream that fits its use.

Visual Analogy

Imagine you have a big painting (sensor image), and:

- You take a **postcard-sized crop** (preview)
- A **poster-sized crop** (video)
- A **full-size copy** (photo)

Each crop comes from the **same original**, just cut out differently.

What's happening internally:

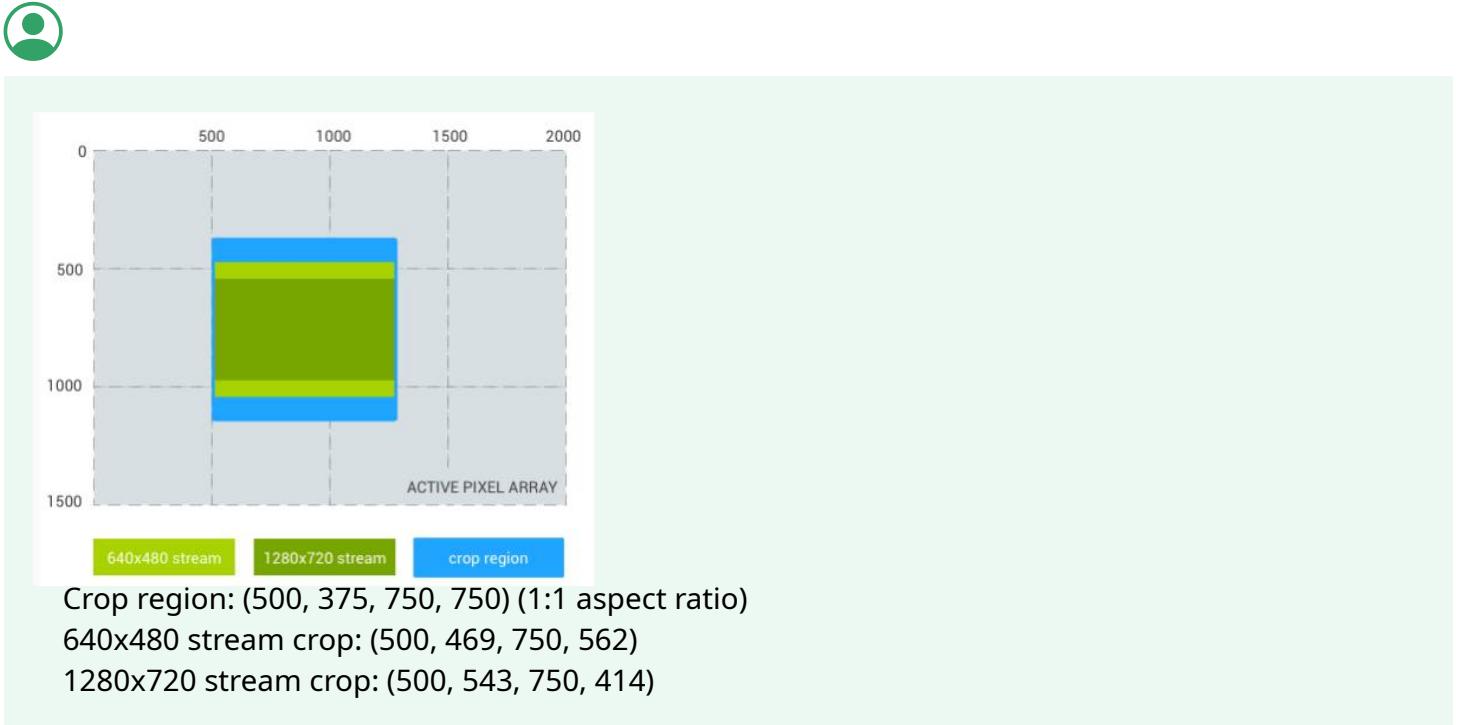
- The camera HAL sets a **main crop region** on the sensor.
- Each stream then takes a **sub-crop** or **scaled version** of that main region to match its resolution and shape.

Summary

Reason for Multiple Crop Streams	Example
Different resolutions	640x480 vs. 1280x720
Different aspect ratios	4:3 (photo) vs. 16:9 (video)
Performance trade-offs	Small preview = less CPU load

Reason for Multiple Crop Streams	Example
Zoom / composition differences	Preview zoom vs. capture zoom

Let me know if you'd like a custom diagram of this — or an animation-style step-by-step.



Great! Let's break down what you're seeing in this image in **simple terms**:

🧠 What this diagram shows:

You have a camera sensor (called the **Active Pixel Array**, the big gray box), and inside that sensor:

- A **blue box** → the main **crop region** selected.
- A **lime green box** → area used by the **640×480 preview stream**.
- A **darker green box** → area used by the **1280×720 video stream**.

■ 1. Crop Region (Blue):

Coordinates: (500, 375) to (1250, 1125)

This is a **750×750 square** (1:1 aspect ratio).

👉 This is the **main crop** of the sensor image — like zooming into the middle square portion of the full sensor.

2. 640×480 Stream Crop:

Aspect Ratio: 4:3 (640×480)

Inside the square crop region, the camera driver takes a **smaller rectangle** (lime green) that fits the 4:3 shape and scales it down to 640×480 for preview.

- **Why offset from top and bottom?**

Because 4:3 is **taller** than 16:9, it fits height-wise into the 1:1 square, leaving extra space on the sides.

3. 1280×720 Stream Crop:

Aspect Ratio: 16:9 (wider than tall)

Another crop is made from the same blue square — this time with a **16:9 shape** (dark green box), which fits side-to-side, leaving extra space on top and bottom.

What's the point?

Even though **both streams** use the **same main sensor crop** (blue), each one takes a **differently shaped slice** of that crop, depending on:

- Its aspect ratio (4:3 or 16:9)
- The resolution it needs (e.g. 640×480, 1280×720)

Analogy

Think of the sensor as a **big cake** (gray), the crop region as a **square slice** (blue), and the two green boxes as:

- One person wants a **tall rectangle slice** (4:3)
- One wants a **wide rectangle slice** (16:9)

Both slices come from the same blue square!

Would you like me to annotate the image with arrows and labels to show this better?



So streams can't go beyond the crop region?

Exactly — **streams cannot go beyond the crop region.**

Let me explain it simply and clearly:

Crop Region = "Visible Sensor Area"

When you set a `crop region` in the camera pipeline:

- It defines the **active part** of the camera sensor you're using.
- Any camera **stream (preview, video, etc.) must be fully contained inside** this region.

Think of the Crop Region as a Window

Imagine the sensor is a **big painting**, and you're looking at it through a **square window** (crop region).

- You can only **see and extract** what's inside that window.
- If a stream (e.g. 1280×720) needs an image, it must take it from **within that window**.
- **No stream can see outside** the blue crop box.

Why is this done?

1. **Performance:** Only process part of the sensor data.
2. **Zoom:** Changing the crop region gives a zoom effect.
3. **Consistency:** All output streams align with the same scene framing.

In summary:

-  **Crop region defines the allowed area** to read from the sensor.
-  **Streams cannot go outside** that crop region.
-  Each stream picks a **sub-rectangle inside the crop**, matching its aspect ratio.

Would you like a zoom-level example or how this affects the field of view?



How can a crop region size be affected?

Great question!

The **crop region size** is affected mainly by **zoom level**, **sensor capabilities**, and sometimes **user requests** or **3A algorithms** (like face tracking). Here's a breakdown in simple terms:

1. Zoom level (digital zoom)

- **Zoom in** → crop region becomes **smaller** (you're cropping into the center).
- **Zoom out** → crop region becomes **larger** (you use more of the sensor).

 Example:

Zoom level	Crop region
1.0x (no zoom)	Full sensor area
2.0x (2x zoom)	Center 1/4 of the sensor
4.0x (4x zoom)	Center 1/16 of the sensor

2. Sensor resolution & aspect ratio

- The **physical size** of the sensor (e.g. 4000×3000) limits the maximum crop size.
- The **aspect ratio** (e.g. 16:9, 4:3, 1:1) defines the crop region shape.
- To keep the output aspect ratio correct, the crop region is resized proportionally.

3. Face / object tracking (auto-adjusted)

- In advanced camera apps or auto-tracking use-cases, the **camera may move the crop region** to keep a face or object centered.
- The size may remain fixed, but the **position of the crop region shifts** over the active array.

4. User app settings

- Some apps request a **custom zoom** or **manual crop** using the `SCALER_CROP_REGION API`.
- Developers can directly set crop rectangle (position + size) via `Camera2 API`.

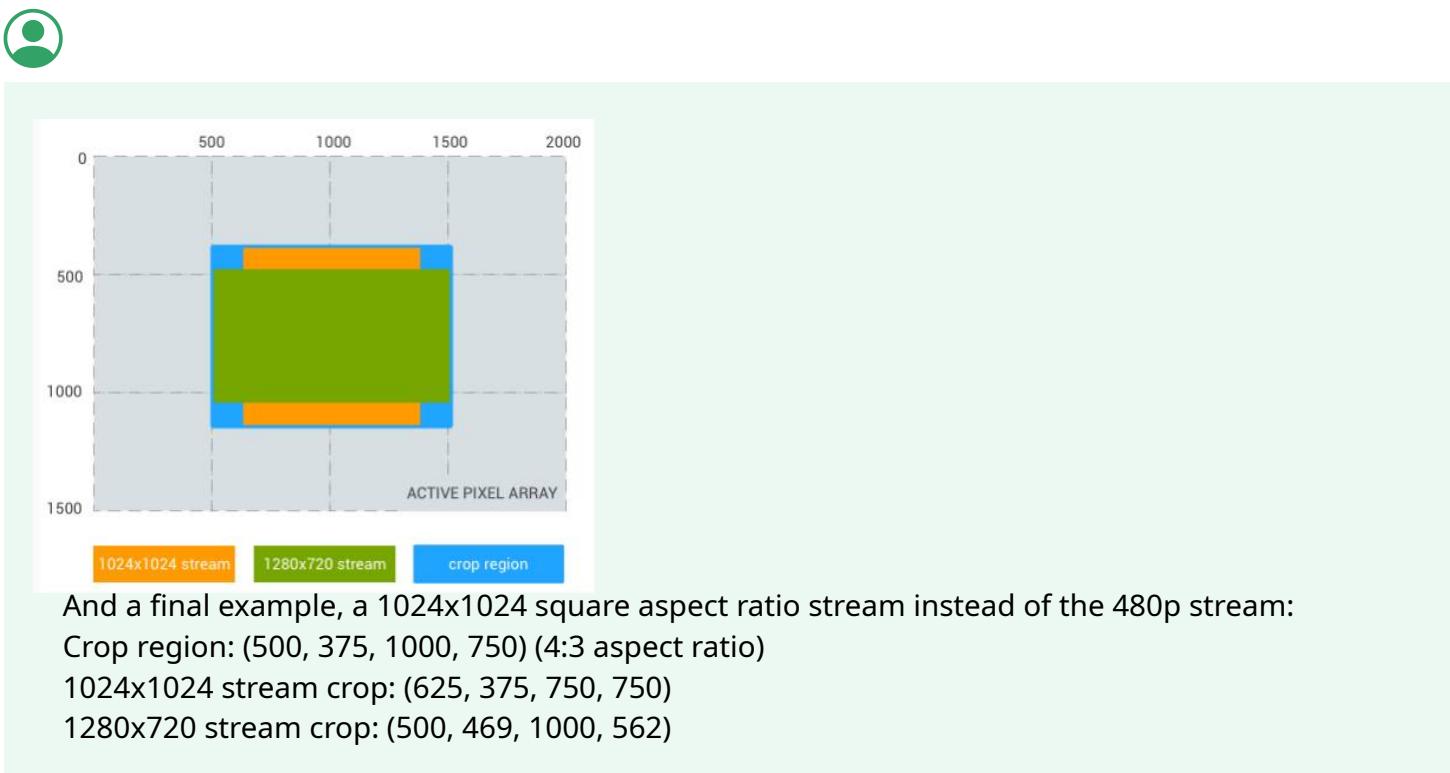
5. Hardware constraints

- The camera **ISP hardware** might support only certain sizes or alignments (e.g., must be divisible by 4 or 8).
- These constraints affect the possible crop region sizes.

✓ Summary

Factor	Effect on Crop Region
Digital zoom	Shrinks size (zooms in)
Full-frame use	Crop = full sensor size
Custom aspect ratio	Crop is shaped to fit (e.g. 16:9, 1:1)
Face tracking	Position shifts, size may stay
App-defined crop	Developer sets it explicitly

Want a live calculation example? I can show how a crop region is calculated from zoom and aspect ratio.



Great! Now let's bring it all together and **clearly explain what's happening** with crop regions and streams using this final example.

🔍 What's going on in the diagram?

- **Active Pixel Array** (the gray area):

This is the full image the camera sensor can see and capture. Think of it as the whole canvas.

- **Crop Region** (blue box):

This is a rectangular area *within* the active pixel array. The camera will only use this region to generate outputs (streams). It defines the "field of view" — what part of the sensor gets sent downstream.

- **Streams** (orange and green boxes):

These are image outputs (of different sizes and shapes) that the app requested — for example:

- Preview
- Video recording
- Still image

Key points in this example:

Item	Value (x, y, width, height)	Aspect Ratio	Notes
Crop Region (blue)	(500, 375, 1000, 750)	4:3	Full field of view — all outputs must fit inside this
1280x720 Stream (green)	(500, 469, 1000, 562)	16:9	Horizontal rectangle (wider) — letterboxed to fit inside crop
1024x1024 Stream (orange)	(625, 375, 750, 750)	1:1	Square — narrower, vertically centered

Answering your core questions

Can streams go outside the crop region?

No.

- All output streams (preview, video, etc.) are **sub-regions** inside the current crop region.
- Crop region acts like a "window" — outputs can only see inside it.

What affects the crop region size?

Several things:

1. Zoom level:

- Higher zoom = smaller crop region = tighter view.

2. App request via `CaptureRequest.SCALER_CROP_REGION`:

- The app or framework sets this value.

3. Camera HAL behavior:

- The HAL chooses a crop region that works for *all requested output streams*.
- It ensures all output streams can fit (in different shapes) inside the same crop.

Why have a crop region at all?

- To control **zoom** and **field of view** without moving the lens.
- To ensure **all output streams** (even with different aspect ratios) come from a consistent part of the image.
- To maintain proper **sensor alignment** and **scaling efficiency** in the camera hardware pipeline.

If you'd like, I can also walk through how **digital zoom** works using crop regions or how **multiple aspect ratios are fit inside the same crop**.



Reprocessing

Additional support for raw image files is provided by reprocessing support for RAW Bayer data. This support allows the camera pipeline to process a previously captured RAW buffer and metadata (an entire frame that was recorded previously), to produce a new rendered YUV or JPEG output.

Zoom

For devices running Android 11 or higher, an app can use a camera's zoom (digital and optical) through the `ANDROID_CONTROL_ZOOM_RATIO` setting.

The zoom ratio is defined as a floating point factor. Instead of using `ANDROID_SCALER_CROP_REGION` for crop and zoom, an app can use `ANDROID_CONTROL_ZOOM_RATIO` to control the zoom level, and use `ANDROID_SCALER_CROP_REGION` for horizontal and vertical cropping to achieve aspect ratios different than the native camera sensor.

A multi-camera system may contain more than one lens with different focal lengths, and the user can use optical zoom by switching between lenses. Using `ANDROID_CONTROL_ZOOM_RATIO` has benefits in the scenarios below:

Zooming in from a wide lens to a telephoto lens: A floating point ratio provides better precision compared to integer values of `ANDROID_SCALER_CROP_REGION`.

Zooming out from a wide lens to an ultrawide lens: `ANDROID_CONTROL_ZOOM_RATIO` supports zoom-out (<1.0f) whereas `ANDROID_SCALER_CROP_REGION` doesn't.

To illustrate, here are several scenarios of different zoom ratios, crop regions, and output streams, using the same hypothetical camera defined in the previous section.

Let's break this down in simple terms:

What is Reprocessing in Camera?

- **Reprocessing** means taking a **RAW image** (the original unprocessed sensor data) that was already captured **earlier**, and **processing it again** through the camera pipeline.
 - This allows you to create a **new JPEG or YUV** version later — maybe with a different white balance, zoom, or tone mapping.
 - This is useful when:
 - You want to tweak how the image looks **after** it's captured.
 - You want to generate **multiple versions** (e.g., color vs. black and white) from the **same RAW** image.
-

What is Zoom Ratio and How is it Different from Crop Region?

Older Android camera systems used:

- **ANDROID_SCALER_CROP_REGION**: a rectangle on the sensor that crops the image and simulates zoom by selecting a smaller portion of the sensor.

Newer Android systems (Android 11+) also support:

- **ANDROID_CONTROL_ZOOM_RATIO**: a **floating-point number** (e.g., **2.0** = 2x zoom, **0.5** = zoomed out).
-

Benefits of ZOOM_RATIO Over CROP_REGION

Feature	ZOOM_RATIO	CROP_REGION
Precision	High (float, e.g., 1.25x)	Low (integer pixels only)
Zoom Out (<1.0x)	 Supported	 Not supported
Switching lenses (e.g., wide ↔ telephoto)	 Smooth	 Hard to manage
Aspect ratio control	 Only zoom	 Crop into specific ratios

How Do They Work Together?

- Use **ZOOM_RATIO** to smoothly zoom in and out.
- Use **CROP_REGION** to crop **horizontally or vertically**, e.g., to get:
 - A square image (1:1)
 - A wide cinematic shot (16:9)
 - A portrait mode (4:3)

This allows **separation** of:

- **Zoom control** (magnification)

- **Framing control** (what part of the scene to capture)

Summary

- **Reprocessing** = Re-use a previously captured RAW frame and metadata to generate new images.
- **ZOOM_RATIO** = Smooth zoom control, including zoom-out, across multiple cameras.
- **CROP_REGION** = Set custom crop shapes/aspect ratios on top of the zoom.

Let me know if you'd like this visualized with an example zoom + crop setup!



confusing

No problem — let's simplify it even more, step by step.

1. What is Reprocessing?

- Your camera captures a **RAW image** (like a digital negative).
- Later, you can **reprocess** that same image to:
 - Change brightness, contrast, white balance
 - Save it again as JPEG or YUV (processed formats)



Example:
You take a photo in RAW. Later, you reprocess it with different filters — without retaking the shot.

2. What is Zoom in Android Camera?

There are **2 ways to zoom**:

A. Old Way: ANDROID_SCALER_CROP_REGION

- You crop a smaller area from the image.
- That looks like zooming in.
- But:
 - It's not smooth
 - It uses integers (not precise)
 - You **cannot zoom out** beyond the sensor area

B. New Way (Android 11+): ANDROID_CONTROL_ZOOM_RATIO

- You use a **decimal zoom factor** like:
 - **1.0** = no zoom
 - **2.0** = 2× zoom
 - **0.5** = zoomed out
- It works smoothly and supports **zooming out** too.
- It can switch between wide, ultra-wide, and telephoto lenses automatically.

Example Comparison

Zoom Method	Can Zoom Out?	Smooth?	Precision	Multi-lens Switch
Crop Region	 No	 No	 Low	 Manual
Zoom Ratio	 Yes	 Yes	 High	 Automatic

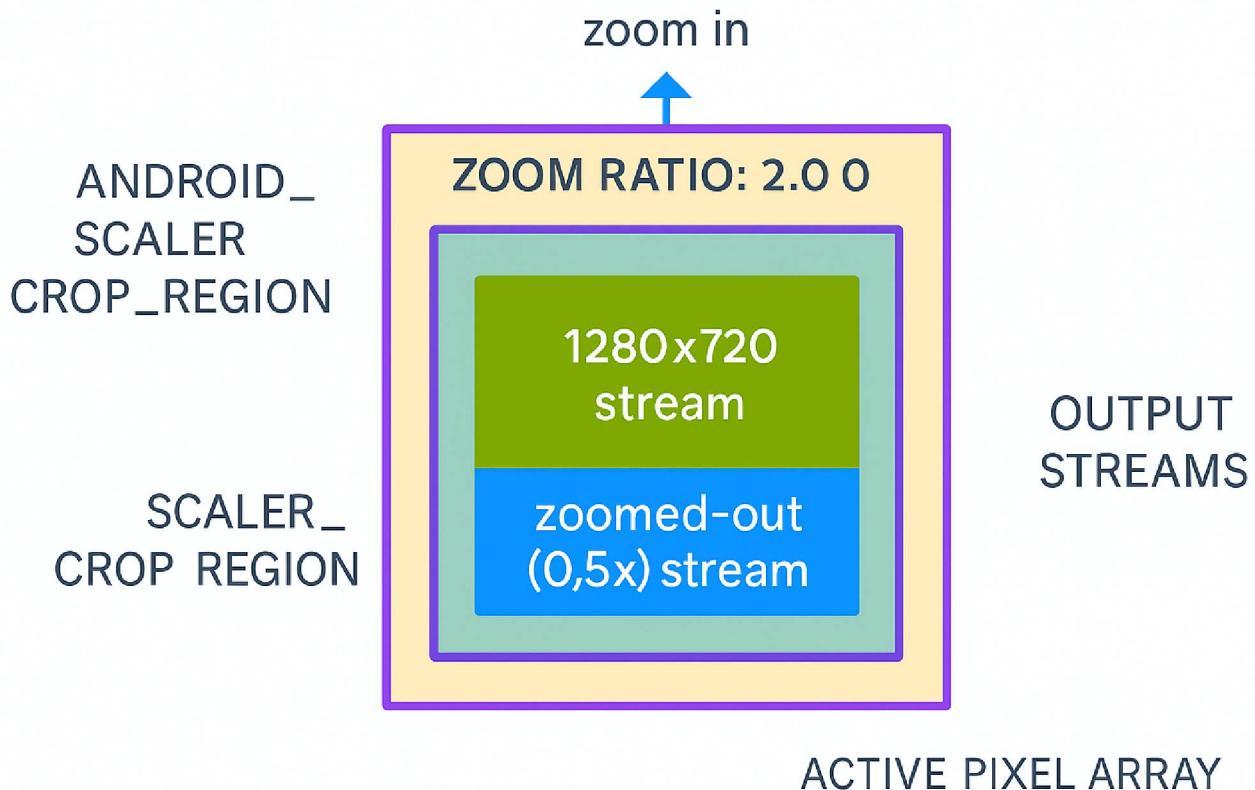
Use Both Together?

Yes! Use:

- **ZOOM_RATIO** for zooming in/out smoothly
- **CROP_REGION** if you want a **different shape** (like square photo)

Would you like a picture example of this?





Request creation and submission

`bookmark_border`

Default requests

To construct default capture requests, call

`ICameraDeviceSession::constructDefaultRequestSettings()`.

Request submission

To submit camera capture requests, call `ICameraDeviceSession::processCaptureRequest()`.

Miscellaneous methods

Request/result message queues

Camera capture result and request IPC overhead can be further optimized with fast message queues. Call the `ICameraDeviceSession::getCaptureRequestMetadataQueue()` and `ICameraDeviceSession::getCaptureResultMetadataQueue()` methods to query the corresponding queues.

Flush requests

To flush any pending capture requests, call `ICameraDeviceSession::flush()`.

Here's a **simple explanation** of each term from your message:

1. Request Creation

- `constructDefaultRequestSettings()`

This creates a **default camera request template**.

- Think of it like: "Give me a basic set of camera settings (like auto focus, exposure, etc.) that I can start with."
- You can modify these settings later before capturing.

2. Request Submission

- `processCaptureRequest()`

This sends your camera request to the hardware.

- It's like: "Now that I've prepared my camera settings, go ahead and take the picture (or start the video stream)."

3. Message Queues (Performance Boost)

- `getCaptureRequestMetadataQueue()`
- `getCaptureResultMetadataQueue()`

These use **Fast Message Queues (FMQ)** for fast and efficient communication between your app and the camera HAL (Hardware Abstraction Layer).

- They reduce **IPC (inter-process communication)** overhead.
- This means faster handling of metadata for:
 - Sending capture settings → request queue
 - Receiving capture results → result queue

4. Flush Requests

- `flush()`

Cancels all in-progress or queued requests.

- Useful if you need to **stop all camera operations quickly**, like when your app is closing or changing modes.

Would you like a **diagram** showing how this flow works step-by-step?



Stream configurations

bookmark_border

Android includes features allowing camera clients to choose optimal camera streams for specific use cases and to ensure that certain stream combinations are supported by the camera device. A stream configuration refers to a single camera stream configured in the camera device and a stream combination refers to one or more sets of streams configured in the camera device. For more on these features, see recommended stream configurations and API to query feature combinations.

Reference implementation

There is a vendor-side reference implementation of the recommended configuration streams and the API to query stream combination features. You can find this implementation at `QCamera3HWI.cpp`

Recommended stream configurations

Camera vendors can advertise recommended stream configurations for specific use cases to camera clients. These recommended stream configurations, which are subsets of `StreamConfigurationMap`, can help camera clients choose optimal configurations.

Although `StreamConfigurationMap` provides exhaustive stream configuration information to camera clients, it doesn't provide any information about the efficiency, power, or performance impacts of choosing one stream over another. Camera clients can freely choose from all the possible stream configurations but in many cases, this leads to clients using sub-optimal camera configurations and apps making time-consuming exhaustive searches.

For example, although some processed YUV formats are required and must be supported, the camera device might not have native support for the formats. This results in an additional processing pass for the format conversion and reduces efficiency. The size and corresponding aspect ratio can also have a similar impact making particular dimensions preferable in terms of power and performance.

Your recommended stream configuration maps aren't required to be exhaustive compared to the `StreamConfigurationMap`. The suggested configuration maps must follow the requirements in the implementation section and can include any of the available formats, sizes, or other values found in `StreamConfigurationMap`. Hidden formats, sizes, or other values not found in `StreamConfigurationMap` can't be included in recommended stream configuration maps.

All tests remain unchanged and aren't relaxed depending on the recommended stream configurations.

The recommended stream configurations provided by the camera implementation are optional

and the camera client can ignore them.

Implementation

Follow these steps to implement this feature.

Metadata entries

To enable this feature the Camera HAL must populate the following static metadata entries:

`android.scaler.availableRecommendedStreamConfigurations`: The recommended subset of stream configurations for specific use cases. The declaration uses bitmaps indicating the suggested use cases in the form of [1 << PREVIEW | 1 << RECORD..]. The use cases extend the regular (format, width, height, input) tuple with one additional entry. Nonexisting public use cases or any other bits set within the range [PUBLIC_END, VENDOR_START] are prohibited.

This information is stored in the `availableRecommendedStreamConfigurations` metadata tag.

The following example shows an array for a recommended stream configuration for a camera device that only supports 4K and 1080p, where both resolutions are preferred for video recording but only 1080p is suggested for preview.

```
[3840, 2160, HAL_PIXEL_FORMAT_IMPLEMENTATION_DEFINED,
ANDROID_SCALER_AVAILABLE_STREAM_CONFIGURATIONS_OUTPUT,
(1 << ANDROID_SCALER_AVAILABLE_RECOMMENDED_STREAM_CONFIGURATIONS_RECORD |
1 << ANDROID_SCALER_AVAILABLE_RECOMMENDED_STREAM_CONFIGURATIONS_SNAPSHOT |
1 <<
ANDROID_SCALER_AVAILABLE_RECOMMENDED_STREAM_CONFIGURATIONS_VIDEO_SNAPSHOT)

,
1920, 1080, HAL_PIXEL_FORMAT_IMPLEMENTATION_DEFINED,
ANDROID_SCALER_AVAILABLE_STREAM_CONFIGURATIONS_OUTPUT,
(1 << ANDROID_SCALER_AVAILABLE_RECOMMENDED_STREAM_CONFIGURATIONS_PREVIEW |
1 << ANDROID_SCALER_AVAILABLE_RECOMMENDED_STREAM_CONFIGURATIONS_RECORD |
1 << ANDROID_SCALER_AVAILABLE_RECOMMENDED_STREAM_CONFIGURATIONS_SNAPSHOT |
1 <<
ANDROID_SCALER_AVAILABLE_RECOMMENDED_STREAM_CONFIGURATIONS_VIDEO_SNAPSHOT)
]
```

Note: The above array example only illustrates one particular scenario and isn't complete. The complete final array must include the recommended stream configurations for all required use cases mentioned below.

`android.depth.availableRecommendedDepthStreamConfigurations` (available only if supported by device): The recommended depth dataspace stream configurations suggested for this camera device. Similar to the above metadata entry, an additional use case bitmap indicates the suggested use cases.

This information is stored in the `availableRecommendedInputOutputFormatsMap` metadata tag.

`android.scaler.availableRecommendedInputOutputFormatsMap` (available only if supported by device): The mapping of recommended image formats that are suggested for this camera device for input streams, to their corresponding output formats.

This information is stored in the `availableRecommendedDepthStreamConfigurations` metadata tag.

This information is available to camera clients through the `RecommendedStreamConfigurationMap` API.

Required use cases

Recommended stream configurations must be provided for the following use cases and meet the corresponding requirements:

Use case	Requirement
----------	-------------

`PREVIEW` A preview must only include nonstalling processed stream configurations with output formats such as `YUV_420_888` and `IMPLEMENTATION_DEFINED`.

`RECORD` A video record must include stream configurations that match the advertised supported media profiles with the `IMPLEMENTATION_DEFINED` format.

`VIDEO_SNAPSHOT` A video snapshot must include stream configurations that are at least as large as the maximum `RECORD` resolutions and only with the `BLOB + DATASPACE_JFIF` format/dataspace combination (JPEG). The configurations shouldn't cause preview glitches and should be able to run at 30 fps.

`SNAPSHOT` Snapshot stream configurations must include at least one with a size close to `android.sensor.info.activeArraySize` with the `BLOB + DATASPACE_JFIF` format/dataspace combination (JPEG). Taking into account restrictions on aspect ratio, alignment, and other vendor-specific restrictions, the area of the maximum suggested size shouldn't be less than 97% of the sensor array size area.

`ZSL` (if supported) If supported by the camera device, recommended input stream configurations must only be advertised together with other processed or stalling output formats.

`RAW` (if supported) If supported by the camera device, recommended raw stream configurations must only include RAW based output formats.

Other use cases

You can provide additional recommended configuration streams for use cases specific to your implementation.

Validation

To test your implementation of the recommended configuration streams, run the following CTS and VTS tests:

CTS: `ExtendedCameraCharacteristicsTest.java`

VTS: `VtsHalCameraProviderV2_4TargetTest.cpp`

API to query feature combinations

Starting in Android 15, the Android platform provides an API to query feature combinations. This API lets camera clients query whether a specified combination of features can be supported by the device. This API is necessary because the camera2 API models different features such as 4k, 60fps, HDR video, UltraHDR, Ultrawide zoom, and stabilization as orthogonal controls.

Requirements

To support the API to query feature combinations, the camera HAL must implement version 3 of the ICameraDevice interface. For details, see the Implementation section.

When the API is supported, preview stabilization must be orthogonal to other features. This means that for a camera device that supports preview stabilization, the return value of `isStreamCombinationWithSettingsSupported` for a particular combination must be the same value when preview stabilization is on or off. This reduces the search space for feature combination queries.

In addition, for media performance class 15, the primary rear camera must support preview stabilization with 10-bit HLG10 preview for 1080p and 720p preview and maximum size JPEGs. For more details on these requirements, see section 2.2.7.2. Camera of the CDD.

Implementation

To support the API to query feature combinations, implement the following feature combination query APIs in version 3 of ICameraDevice:

`constructDefaultRequestSettings`: Creates the default settings for a specified `CaptureRequest` type. The HAL can use the `ICameraDeviceSession::constructDefaultRequestSettings` implementation.

`isStreamCombinationWithSettingsSupported`: Checks for device support of a specified camera stream combination with session parameters and additional `CaptureRequest` keys. Must return true for supported combinations and false for unsupported feature combinations. The HAL can use the `isStreamCombinationSupported` implementation and add support to check the `CaptureRequest` settings being passed in `sessionParams`.

`getSessionCharacteristics`: Takes a supported stream combination with session parameters and returns the session-specific characteristics.

`INFO_SESSION_CONFIGURATION_QUERY_VERSION`: Lists all the commonly used session configurations. These configurations are verified by compliance testing.

For versions lower than version 3 of the ICameraDevice interface, the HAL should implement the `isStreamCombinationSupported` method.

For more information on the feature combinations queried by the API, see the documentation on `sessionConfigurationQueryVersion` in `system/media/camera/docs/metadata_definitions.xml`.

For a reference implementation of this feature, see
<hardware/google/camera/devices/EmulatedCamera/hwl/>.

Public APIs

Apps can use the following public APIs to query supported feature combinations for the device:

`CameraDevice.CameraDeviceSetup`: A limited representation of `CameraDevice` that can be used to query feature combinations without requiring a `CameraDevice` instance.

`getCameraDeviceSetup`: Acquires a `CameraDeviceSetup` object for a given camera ID if `isCameraDeviceSetupSupported` returns true.

`INFO_SESSION_CONFIGURATION_QUERY_VERSION`: Supports feature combination queries if this value is `VANILLA_ICE_CREAM` or higher.

`OutputConfiguration`: A class describing camera output, which can contain a deferred surface for the purpose of low latency feature combination queries.

`SessionConfiguration`: A utility class describing the session configuration including stream combinations and session parameters, which can be used for feature combination queries.

Validation

To validate your implementation of this feature, use the following VTS, CTS, and Camera ITS (CTS Verifier) tests:

VTS

`hardware/interfaces/camera/provider/aidl/vts/VtsAidlHalCameraProvider_TargetTest.cpp`
`CTS`

`cts/tests/camera/src/android/hardware/camera2/cts/FeatureCombinationTest.java`
`cts/tests/camera/src/android/hardware/camera2/cts/CameraDeviceSetupTest.java`
`Camera ITS`

`cts/apps/CameraITS/tests/feature_combination/test_feature_combination.py`
`cts/apps/CameraITS/tests/scene6/test_session_characteristics_zoom.py`

Here's a **simplified summary** of all the info about **stream configurations and feature combinations** in the Android Camera HAL:

What is a Stream Configuration?

- A **stream** is how the camera delivers image data (e.g., to preview, record, save, etc.).

- A **stream configuration** is one setup of a stream: its **format** (like YUV, JPEG), **size** (e.g., 1920x1080), and **use case** (like preview, video, etc.).
-

What is a Stream Combination?

- A **stream combination** is when **multiple streams are active at once**, like:
 - Preview (for display)
 - Video recording (for saving)
 - Snapshot (for still photos during video)

Example: You might have:

- 1280x720 YUV for **preview**
 - 1920x1080 IMPLEMENTATION_DEFINED for **recording**
 - JPEG for **video snapshot**
-

Why Do We Need Recommended Stream Configurations?

- Some stream setups are **better** than others (more power-efficient, faster, etc.).
- The camera HAL can **recommend optimal configurations** for:
 - PREVIEW
 - RECORD
 - SNAPSHOT
 - VIDEO_SNAPSHOT
 - ZSL (Zero Shutter Lag)
 - RAW

These are provided in:

- `android.scaler.availableRecommendedStreamConfigurations`
- And similar metadata for depth and input/output maps.

Apps can use these to avoid **trial and error** or **slow setups**.

How Can Apps Check Which Features Work Together?

Starting in **Android 15**, apps can now check:

"Does this stream + format + resolution + effect (e.g., HDR or stabilization) work on this device?"

 This is done using:

- `isStreamCombinationWithSettingsSupported()`
- `constructDefaultRequestSettings()`
- `getSessionCharacteristics()`

These let apps avoid surprises (like features not working together).

Examples of Use Cases

Use Case	Must Include
PREVIEW	Fast YUV or IMPLEMENTATION_DEFINED
RECORD	Matches video profiles (e.g., 1080p)
SNAPSHOT	Full-size JPEG, 97%+ sensor size
VIDEO_SNAPSHOT	At least as large as RECORD, JPEG
RAW	RAW formats only
ZSL	Input + output combo if supported

Tools & APIs for Developers

Public Android APIs:

- `CameraDeviceSetup`: Lets apps check feature combos **without opening the full camera**
- `getCameraDeviceSetup`: Gets setup object for a camera ID
- `SessionConfiguration & OutputConfiguration`: Describes stream layouts

How to Validate

Google provides tests to check if HAL implementation follows the rules:

- **CTS** (Compatibility Test Suite)
- **VTS** (Vendor Test Suite)
- **Camera ITS** (Image Test Suite for camera quality)

Would you like a **visual flowchart** or **example JSON** of a stream configuration map to make this even clearer?



* **Camera ITS** (Image Test Suite for camera quality)
explain more

Certainly! Let's break down **Camera ITS (Image Test Suite)** in the context of Android Camera HAL testing — what it is, why it exists, how it works, and how it ties in with stream configuration testing.

What is Camera ITS?

Camera ITS stands for **Camera Image Test Suite**.

It's part of the **CTS Verifier**, designed to validate that **camera image quality and behavior** meets the standards required by Android compatibility.

What Does Camera ITS Test?

It tests **real camera behavior and output**, such as:

Category	What It Tests
Image Quality	Sharpness, noise, exposure, white balance
3A (Auto)	Auto-focus (AF), Auto-exposure (AE), Auto-white-balance (AWB) accuracy
Stream Configuration	Whether required and recommended stream formats/resolutions behave correctly
Format Consistency	RAW vs YUV vs JPEG alignment
Sensor Properties	Noise model, gain linearity, color shading
ZSL Support	Zero-shutter-lag stream capture behavior

How Are Tests Run?

ITS is designed to be run in **lab conditions** with:

- A **robot-controlled test rig** (for consistent movement and alignment)
- **Scene test charts** (like ISO resolution charts, color checker boards)
- A Python test framework
- A test Android app (`com.android.cts.verifier`) to control the camera

Components of ITS

1. Python Scripts (host side)

- Run from your Linux/macOS PC
- Sends commands to the device and parses images

2. ITS App (on device)

- Talks to Android's Camera2 APIs
- Receives capture requests
- Sends back captured images and metadata

3. Scene Setup

- Uses printed test charts, LED lights, controlled lighting, etc.
-

Where Does Stream Configuration Fit In?

Camera ITS verifies that:

- The **recommended stream configurations** listed in `camera_characteristics` actually work.
- The combinations **do not crash** or cause **unexpected results**.
- ZSL input + output works **with guaranteed latency**.
- JPEG/YUV stream resolution **matches expectations** (e.g., >97% of full sensor size).
- **Aspect ratio and scaling** match what the HAL reports.

It helps catch bugs like:

- Wrong cropping behavior
 - Preview output not matching JPEG frame
 - JPEGs missing metadata or too slow
 - Stream combination crashing the HAL
-

How ITS Complements CTS/VTS

Suite	Focus
CTS	Java API conformance (e.g., Camera2 APIs return expected values)
VTS	HAL interface (HIDL/AIDL) compliance at the binder level
ITS	Image output validation , actual camera behavior under real-world tests

⚠️ Is ITS Mandatory?

- Yes, for devices with **Camera HAL Level 3** or higher.
- **Mandatory for GMS certification** for most phones with advanced cameras.
- Even for **Limited** or **Full** HAL level devices, some ITS tests may still be required if claiming support for certain features.

🔧 Example: How Stream Configuration Is Tested in ITS

Let's say your device lists this in `availableStreamConfigurations`:

- YUV_420_888 @ 1280x720
- IMPLEMENTATION_DEFINED @ 1920x1080
- JPEG @ 4000x3000

ITS will:

1. Submit a **session config** with all three streams active
2. Send test patterns to preview and compare JPEG output
3. Check metadata (e.g., AE/AF lock consistency)
4. Measure latency and correctness
5. Fail the test if:
 - Stream drops
 - Metadata is inconsistent
 - Preview diverges from JPEG output

Would you like:

- A visual setup of the ITS camera test rig?
- A sample ITS test log (e.g., from `test_yuv_plus_jpeg`)?
- Or explanation of how to run ITS locally using Pixel or Cuttlefish?



is request come under stream

Great question — let's clarify how **requests** and **streams** relate in Android's **Camera HAL (Hardware Abstraction Layer)** architecture.

Summary of Terms

- **Stream:** A continuous flow of image buffers (e.g., preview, video, JPEG).
- **Capture Request:** A one-time command telling the camera how to capture a frame, what settings to use, and where to send the image buffers (i.e., which streams).

Does a request come under a stream?

No, it's the **other way around**:

 A **request references one or more streams** — it tells the camera:

“Capture a frame with these settings, and send the output to this list of streams.”

So, a **request uses streams**. The request contains:

- Settings (exposure, ISO, crop, zoom, focus, etc.)
- Input buffer (if using reprocessing)
- Output buffer targets → mapped to active **streams**

How they fit together

1. Define streams first

When the camera is initialized, the app (via Camera HAL) sets up **output streams**, e.g.:

- Stream 0 → 1280x720 YUV for preview
- Stream 1 → 640x480 JPEG for photo
- Stream 2 → 1024x1024 YUV for special cropping

These are defined during the `configureStreams()` step.

2. Submit capture requests

Then the app sends **requests** using `processCaptureRequest()`, e.g.:

```
cpp

CaptureRequest {
    settings = { zoomRatio=2.0, AE=on, AF=continuous, cropRegion=... }
    outputStreams = [ stream 0, stream 1 ] // preview + capture
}
```

Each request can:

- Use 1 or more streams
- Target different resolutions
- Apply crop, zoom, and other tuning parameters

Analogy

Think of it like printing:

- **Streams = different printers** (photo, A4, PDF)
- **Request = print job** (content + settings + which printer to use)

Fast Message Queues

To reduce IPC overhead (since request/result data are exchanged often):

- `getCaptureRequestMetadataQueue()` → faster metadata delivery from app to HAL
- `getCaptureResultMetadataQueue()` → faster metadata return from HAL to app

This avoids Binder overhead for each capture.

Summary

Concept	Description
Stream	Output image target (resolution + format), declared during configuration
Capture Request	A per-frame command: which streams to use + how to capture
Relation	Requests reference streams — they don't belong <i>under</i> streams

Let me know if you'd like an illustrated flow or code example!



Which IPC mechanism they in camera service to cameraHAL

Excellent question!

In Android, the **CameraService (framework)** communicates with the **Camera HAL (vendor code)** using **HIDL (HAL Interface Definition Language)** or **AIDL (in newer versions)**. These sit on top of the **Binder IPC** mechanism.

IPC Mechanism Between CameraService and Camera HAL

Layer	IPC Mechanism Used	Description
CameraService → HAL (≤ Android 10)	HIDL + Binder IPC	Framework uses HIDL-defined interfaces to call into vendor HAL through Binder
CameraService → HAL (Android 11+)	AIDL + Binder IPC (for newer devices)	Google has begun migrating to AIDL HALs
Metadata + buffers	FMQ (Fast Message Queue) + GRALLOC/ION/DMABUF	Shared memory to reduce Binder overhead for metadata and buffer passing

Bricks Components in IPC Path

◆ 1. CameraService

- In `/frameworks/av/services/camera/libcameraservice/`
- Talks to the HAL via IPC through interfaces like `ICameraDeviceSession`.

◆ 2. CameraProvider

- Acts as a bridge to the HAL, either via:
 - `android.hardware.camera.provider@2.x` (HIDL)
 - `android.hardware.camera.provider` (AIDL)

◆ 3. Binder IPC

- Core Android IPC mechanism
- Used to transfer commands, metadata, and configuration between framework and HAL

◆ 4. Fast Message Queues (FMQ)

- Shared memory ring buffers
- Used for **metadata** (capture settings + results)
- Reduces frequent small Binder calls (like settings per frame)

◆ 5. Gralloc / ION / DMA-BUF

- Used to pass actual image buffers (YUV/JPEG/etc.)
- Shared memory handles passed via Binder to HAL

Flows Typical Flow

1. App → CameraService (Binder)

Java app requests camera access and sends capture requests.

2. CameraService → Camera HAL (Binder + HIDL/AIDL)

CameraService uses HIDL/AIDL interfaces like:

- **ICameraDeviceSession**
- **ICameraProvider**

3. Metadata transfer via FMQ

- **getCaptureRequestMetadataQueue()**
- **getCaptureResultMetadataQueue()**

4. Image buffers shared via gralloc / dmabuf

HIDL Interfaces Example

In Android ≤ 10:

```
cpp

android.hardware.camera.device@3.4::ICameraDeviceSession
android.hardware.camera.provider@2.4::ICameraProvider
```

In Android ≥ 11 (migrating to AIDL):

```
aidl

android.hardware.camera.provider.ICameraProvider
android.hardware.camera.device.ICameraDeviceSession
```

Summary

Channel	Mechanism
Commands	Binder IPC via HIDL/AIDL
Metadata	FMQ (Fast Message Queue)
Image Buffers	Shared memory (gralloc, ION, dmabuf)

Would you like a diagram showing this flow visually, or code references for AIDL vs HIDL interfaces?





Commands**Binder IPC via HIDL/AIDL**Metadata**FMQ (Fast Message Queue)**Image Buffers**Shared memory (gralloc, ION, dmabuf)**
explain in more details

Sure! Let's break this down **layer by layer** and explain in **clear detail** how the **Camera Framework (CameraService)** talks to the **Camera HAL** through different IPC and memory mechanisms:

OVERVIEW: 3 PATHS from CameraService to Camera HAL

Data Type	Mechanism Used	Purpose
Commands / API Calls	Binder IPC via HIDL or AIDL	For setup, open/close, capture/start stream, configure pipeline
Metadata	FMQ (Fast Message Queue)	For fast, zero-copy transmission of capture request and result metadata
Image Buffers	Shared Memory (gralloc, ION, DMA-BUF)	For high-throughput, large-size image buffers like RAW/YUV/JPEG

1. COMMANDS: Binder IPC via HIDL or AIDL

What goes here?

- Open/close camera
- Stream configuration
- Submit capture requests
- Error callbacks
- Flush

Technologies:

- **Binder**: Low-level kernel IPC system in Android
- **HIDL (≤ Android 10)**: Interface definition for HALs
- **AIDL (≥ Android 11)**: Newer version, easier to use, backward compatible

Flow:

nginx

App → CameraService → ICameraProvider → ICameraDevice → Camera HAL

Example:

```
cpp
cameraDevice->configureStreams(streamConfig); // via Binder HIDL call
cameraDevice->processCaptureRequest(request); // via HIDL IPC
```

These calls are **Binder transactions** marshaling data like structs, enums, flags.

2. METADATA: FMQ (Fast Message Queue)

What is metadata?

- Settings for capture (exposure, ISO, white balance)
- Results from HAL (timestamp, 3A stats, lens state)

Problem:

Metadata changes **every frame** (~30+ per second). Sending via Binder is too slow!

Solution: FMQ

Fast Message Queue:

- A zero-copy shared ring buffer in memory
- Shared between CameraService and HAL
- Backed by `ashmem` or `memfd`
- Works like a circular queue

Interfaces:

- `getCaptureRequestMetadataQueue()`
- `getCaptureResultMetadataQueue()`

Flow:

```
sql
```

```
CameraService → Push request metadata to FMQ → HAL reads
HAL → Push result metadata to FMQ → CameraService reads
```

FMQ avoids Binder overhead and allows **real-time metadata exchange**.

3. IMAGE BUFFERS: Shared Memory (gralloc / ION / dma-buf)

What are image buffers?

- Big buffers that store RAW/YUV/JPEG frames
- Each buffer is ~5-15 MB

Why not Binder?

- Binder can't handle large, frequent data.
- Binder has 1MB total per process.

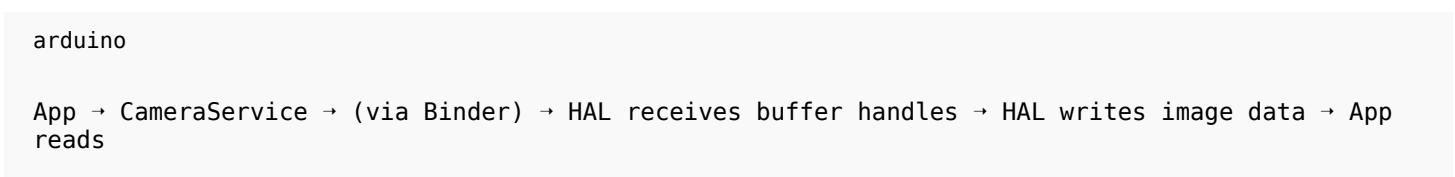
Solution: Shared memory handles using:

Memory Allocator	Used for
gralloc	GPU and displayable buffers
ION (deprecated)	Shared memory allocation (older)
DMA-BUF	Modern zero-copy buffer sharing across subsystems

How it works:

- App requests a buffer (via `ANativeWindow`, `ImageReader`, etc.)
- Buffer is allocated via gralloc
- HAL receives a **buffer handle** (not the data) via Binder
- HAL maps the buffer and writes directly into it
- No data copy!

Buffer Flow:



End-to-End Example (One Frame Capture)

1. App starts camera

```
java
cameraManager.openCamera(...)
```

- Framework (CameraService) uses Binder to talk to HAL

2. Submit capture request

- HAL receives request via Binder HIDL/AIDL
- Metadata is put in FMQ
- Image buffer handle is passed (not image data)

3. HAL processes frame

- Reads metadata from FMQ
- Uses buffer handle to write image data
- Fills result metadata in FMQ

4. Framework gets result + buffer

- Reads result from FMQ
- Sends callback to App with image

Summary Table

Channel	Technology	Purpose
Commands	Binder (HIDL/AIDL)	Setup, control, capture requests
Metadata	FMQ	Fast frame-by-frame settings/result
Buffers	gralloc / dma-buf	Shared access to image memory

Would you like the full source locations (e.g., `ICameraDeviceSession.aidl`, `Camera3Device.cpp`, etc.) or a real-life trace (e.g., `atrace`, `binder dump`) showing this flow?



Camera HAL3 buffer management APIs

bookmark_border

Android 10 introduces optional camera HAL3 buffer management APIs that allow you to implement buffer management logic to achieve different memory and capture latency tradeoffs

in camera HAL implementations.

The camera HAL requires N requests (where N is equal to the pipeline depth) queued in its pipeline, but it often doesn't require all N sets of output buffers at the same time.

For example, the HAL might have eight requests queued in the pipeline, but it only requires output buffers for the two requests in the last stages of the pipeline. On devices running Android 9 and lower, the camera framework allocates buffers when the request is queued in the HAL so there could be six sets of buffers in the HAL that aren't in use. In Android 10, the camera HAL3 buffer management APIs allow for the decoupling of the output buffers to free up the six sets of buffers. This can lead to hundreds of megabytes of memory savings on high-end devices and can also be beneficial for low-memory devices.

Figure 1 shows a diagram of the camera HAL interface for devices running Android 9 and lower. Figure 2 shows the camera HAL interface in Android 10 with the camera HAL3 buffer management APIs implemented.

Let's break this down **in simple terms**, with a **focus on how buffer management changed from Android 9 to Android 10+**.

What is a Camera Request?

A **capture request** is a message sent by the Camera framework to the Camera HAL, saying:

"Please capture a frame with these settings (e.g. exposure, focus), and send the output to these buffers."

What are Buffers?

Buffers are memory blocks where image data is stored (like RAW, YUV, JPEG). For every frame captured, output buffers are needed.

- For example: If the request says "send the result to the preview window," a graphic buffer (e.g. `ANativeWindowBuffer`) is provided.
- The **request contains handles to buffers** for each stream (e.g., preview, JPEG, etc.)

? What is the Problem with Android 9 and lower?

Let's say the camera HAL pipeline can **hold 8 requests** at a time (`pipeline depth = 8`).

- Android 9: When all 8 requests are queued, **all 8 sets of output buffers must be allocated immediately**.

- But the HAL might **only be working on request #7 and #8**, and **requests #1-6 just sit there, holding their buffers unused**.
- **✗ Wasted memory.** Could be hundreds of MB.

✓ What Changed in Android 10+ (HAL3 Buffer Management APIs)?

With the **new HAL3 buffer management APIs**, Android 10 lets HAL do **lazy allocation** of buffers:

- Only **ask for output buffers when the HAL really needs them** (e.g., last 2 stages of pipeline).
- Frees up buffers for other uses while earlier pipeline stages are running.

This is especially useful on:

- High-res sensors (e.g. 64MP)
- Devices with many simultaneous streams (JPEG + preview + raw)
- Low-memory devices

📊 Diagram Comparison

Android 9

```
csharp

[Request 1] -> buffers allocated
[Request 2] -> buffers allocated
...
[Request 8] -> buffers allocated
==> High memory usage (all buffers tied up)
```

📈 Android 10+

```
rust

[Request 1] -> no buffers yet
[Request 2] -> no buffers yet
...
[Request 7] -> buffers allocated
[Request 8] -> buffers allocated
==> Only last 2 requests use memory buffers
```

🧠 Summary of APIs:

API	Purpose
<code>processCaptureRequest()</code>	Submit a capture request with buffer handles.

API	Purpose
<code>constructDefaultRequestSettings()</code>	Create a template (e.g., preview, video) for settings.
<code>getCaptureRequestMetadataQueue()</code>	Optimize IPC (faster metadata transfer to HAL).
<code>getCaptureResultMetadataQueue()</code>	Optimize IPC (faster metadata from HAL).
<code>flush()</code>	Cancel all pending requests quickly.

Would you like a visual diagram of this difference or how the HAL queues the buffers with and without these APIs?



too much confusing

Let's break it down **step by step in simple terms** — especially focusing on **Camera HAL3 buffer management APIs** and how they changed in Android 10.

Problem Before Android 10 (Android 9 and below):

1. The camera HAL (hardware abstraction layer) works like a **pipeline** — multiple capture requests are being processed in stages.
2. For each request, Android **immediately gives the HAL the buffers** (memory to store image data), **even if the buffer isn't needed right away**.
3. This means:
 - If the HAL is processing 8 requests, and only the last 2 actually need buffers at that point, **6 buffers are sitting idle**.
 - This wastes memory. On high-res cameras, each buffer can be **tens of megabytes**.

What Changed in Android 10 (Camera HAL3 buffer management APIs):

1. Android 10 introduced **optional new APIs** to fix this.
2. Now, buffers are **not given up front for every request**.
3. The HAL can **ask for a buffer only when it's needed**, not when the request is queued.
4. This allows:
 - Memory to be used **more efficiently**.
 - Better performance, especially on **low-memory devices** or **multi-camera phones**.

How It Works:

Two Concepts:

- **Request** = a capture job (like "take a photo")
- **Buffer** = memory to hold the photo/video frame result

With HAL3 Buffer Management:

- When a request is queued → no buffer is allocated yet.
- Later, HAL asks: "Hey, I'm ready to write image output, give me a buffer now."
- Framework gives the buffer **only when it's really needed**.

Related APIs:

1. `processCaptureRequest()` → HAL gets capture request from framework.
2. `constructDefaultRequestSettings()` → framework prepares default capture settings.
3. `getCaptureRequestMetadataQueue()` and `getCaptureResultMetadataQueue()` → Fast Message Queues (FMQ) for metadata transfer (fast IPC).
4. `flush()` → cancels in-flight requests and releases buffers quickly.

Benefit Summary:

Android 9 and below	Android 10+ (with HAL3 buffer API)
Buffers allocated early	Buffers allocated only when needed
Memory often wasted	Efficient memory use
High memory pressure	Better for low-memory or multi-cam use

Let me know if you'd like the **flow diagram simplified**, or a **code example** to make it clearer.



So when memory allocation happens for the buffer for the frame and metadata?

Great question! Let's clearly explain **when memory is allocated** for both:

◆ 1. Metadata buffer

This is small data like exposure, focus, timestamp, etc.

When allocated:

- **Immediately** when the **request is created** by the **camera framework**.
- Passed along to HAL via `processCaptureRequest()`.
- This is lightweight, so no special optimization is needed.

◆ 2. Image buffer (frame data)

This is the heavy one — actual image pixels, often hundreds of megabytes for high-res.

On Android 9 and below:

- Image buffers are allocated **before `processCaptureRequest()`** is called.
- HAL gets **request + buffers together**.
- Memory is **reserved even if HAL doesn't need it yet**, leading to waste.

On Android 10+ (with HAL3 Buffer Management APIs):

- `processCaptureRequest()` is called **without actual image buffers**.
- Buffers are allocated **later, only when HAL asks** via `requestStreamBuffer()` or related callback (behind the scenes).
- This happens when the HAL **actually needs to write** to the buffer (closer to image sensor readout or ISP output).

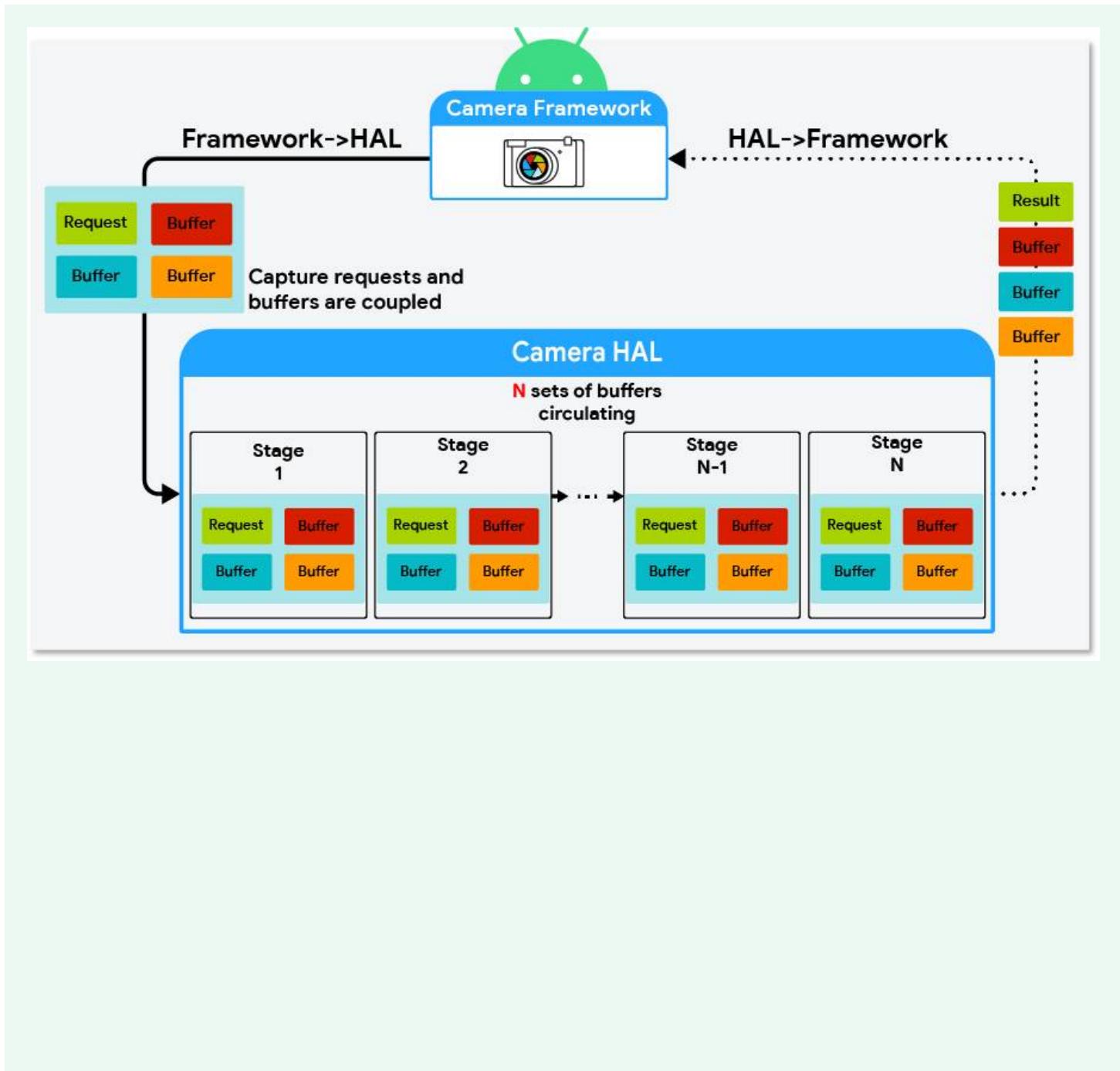
Buffer Flow Summary

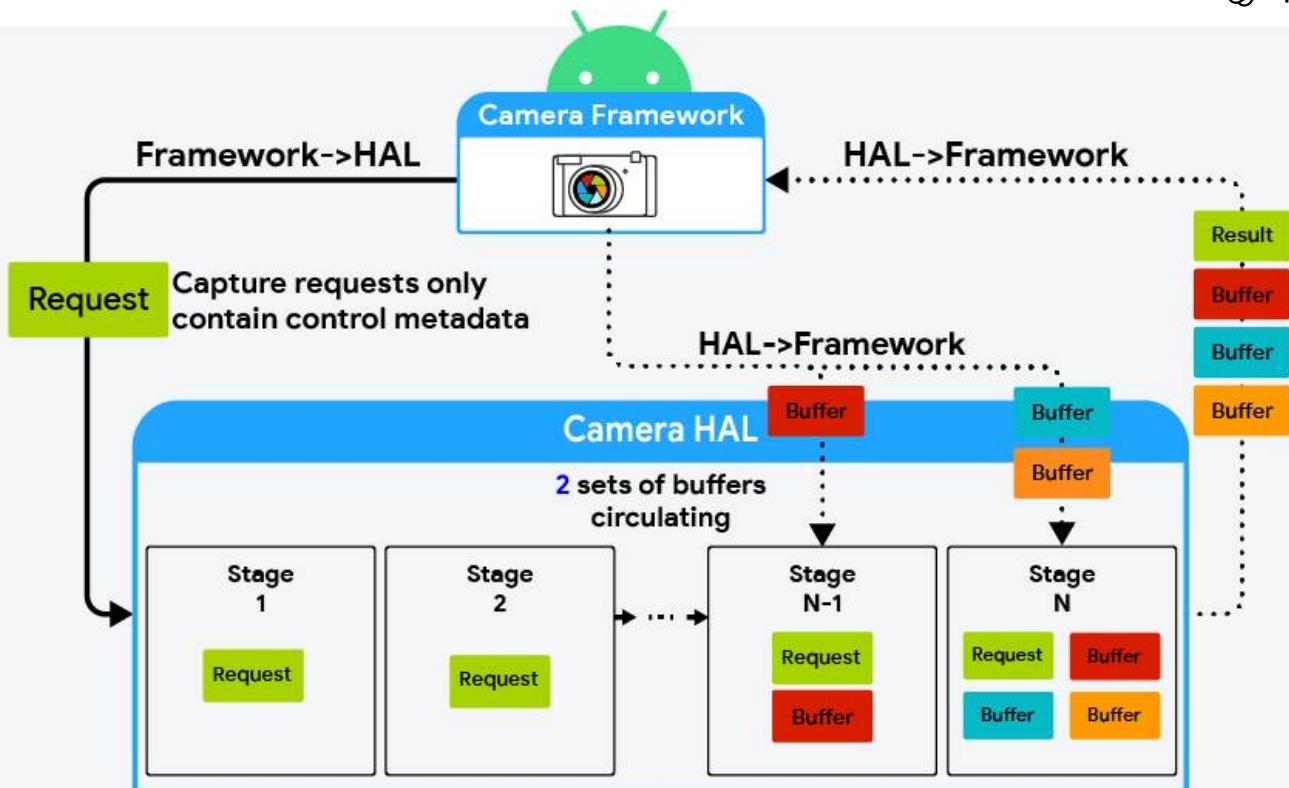
Step	What happens
1. Framework creates capture request	Metadata buffer is created and attached
2. HAL receives request (Android 10+)	Without image buffers
3. HAL processes pipeline stages	Still no buffer needed yet
4. HAL needs to write image output	HAL requests image buffer
5. Framework allocates buffer and gives to HAL	HAL writes the frame
6. HAL sends back <code>capture_result</code> + metadata	App receives result

Why this matters:

- Avoids pre-allocating 6–10 large buffers for requests that are still deep in the pipeline.
- HAL can control how many buffers are active — leading to **better memory performance** and **faster camera startup**.

Would you like to see a **diagram of this flow**, or sample logs or C++ HAL code examples (`camera3_capture_request_t`, `camera3_stream_buffer_t`)?





These two diagrams illustrate **how buffer management in the Camera HAL changed** from Android 9 to Android 10. Let's break them down in simple terms:

Android 9 (First Image) – Tightly Coupled Buffers

- **Each capture request includes both control info and output buffers** (e.g. YUV, JPEG).
- The framework sends N requests *with* buffers to the Camera HAL.
- All N stages in the pipeline hold both the request and its full set of buffers.
- **Problem:** Many buffers (especially early-stage ones) are just sitting idle, using memory.
 - If $N = 8$, but only last 2 need buffers → 6 sets are wasted.
 - On high-res cameras, that's hundreds of MB of RAM!

Android 10 (Second Image) – Decoupled Buffer Management

- Now, **requests contain only control metadata** (e.g. exposure, ISO) when sent to HAL.
- Buffers are sent **later**, just before they are truly needed (typically near the final stage).
- The HAL tracks how far a request has moved in the pipeline, and **only requests the output buffers at the last few stages**.
- **Benefit:**

- Fewer buffers in use at any one time.
- Huge **memory savings** and **lower latency**.

So, when does memory allocation happen now?

- **Before Android 10:** When the request enters the pipeline (even if it's stage 1).
- **Android 10+ with buffer management API:** Memory (buffers) are allocated **only near the end of the pipeline**, when needed.

Would you like a visual explanation of how requests move through the pipeline with this change?



Implement the buffer management APIs

To implement the buffer management APIs, the camera HAL must:

Implement HIDL ICameraDevice@3.5.

Set the camera characteristics key android.info.supportedBufferManagementVersion to HIDL_DEVICE_3_5.

The camera HAL uses the requestStreamBuffers and returnStreamBuffers methods in ICameraDeviceCallback.hal to request and return buffers. The HAL must also implement the signalStreamFlush method in ICameraDeviceSession.hal to signal the camera HAL to return buffers.

Note: If the camera HAL implements these buffer management APIs on a device running Android 9 and lower, the HAL is required to support the API contract illustrated in Figure 1. requestStreamBuffers

Use the requestStreamBuffers method to request buffers from the camera framework. When using the camera HAL3 buffer management APIs, capture requests from the camera framework don't contain output buffers, that is, the bufferId field in StreamBuffer is 0. Therefore, the camera HAL must use requestStreamBuffers to request buffers from the camera framework.

The requestStreamBuffers method allows the caller to request multiple buffers from multiple output streams in a single call, allowing for fewer HIDL IPC calls. However, calls take more time when more buffers are requested at the same time and this might negatively affect the total request-to-result latency. Also, because calls into requestStreamBuffers are serialized in the camera service, it's recommended that the camera HAL use a dedicated high-priority thread to request buffers.

If a buffer request fails, the camera HAL must be able to properly handle nonfatal errors. The following list describes common reasons that buffer requests fail and how they should be handled by the camera HAL.

App disconnects from the output stream: This is a nonfatal error. The camera HAL should send `ERROR_REQUEST` for any capture request targeting a disconnected stream and be ready to process subsequent requests normally.

Timeout: This can occur when an app is busy doing intensive processing while holding onto some buffers. The camera HAL should send `ERROR_REQUEST` for capture requests that can't be fulfilled due to a timeout error and be ready to process subsequent requests normally.

Camera framework is preparing a new stream configuration: The camera HAL should wait until the next `configureStreams` call is complete before calling `requestStreamBuffers` again.

The camera HAL has reached its buffer limit (the `maxBuffers` field): The camera HAL should wait until it returns at least one buffer of the stream before calling `requestStreamBuffers` again.

`returnStreamBuffers`

Use the `returnStreamBuffers` method to return extra buffers to the camera framework. The camera HAL normally returns buffers to the camera framework through the `processCaptureResult` method, but it can only account for capture requests that have been sent to the camera HAL. With the `requestStreamBuffers` method, it's possible for the camera HAL implementation to retain more buffers than what has been requested by the camera framework. This is when the `returnStreamBuffers` method should be used. If the HAL implementation never holds more buffers than requested, the camera HAL implementation doesn't need to call the `returnStreamBuffers` method.

`signalStreamFlush`

The `signalStreamFlush` method is called by the camera framework to notify the camera HAL to return all buffers at hand. This is normally called when the camera framework is about to call `configureStreams` and must drain the camera capture pipeline. Similar to the `returnStreamBuffers` method, if a camera HAL implementation doesn't hold more buffers than requested, it's possible to have an empty implementation of this method.

After the camera framework calls `signalStreamFlush`, the framework stops sending new capture requests to the camera HAL until all buffers have been returned to the camera framework. When all buffers are returned, the `requestStreamBuffers` method calls fail, and the camera framework can continue its work in a clean state. The camera framework then calls either the `configureStreams` or `processCaptureRequest` method. If the camera framework calls the `configureStreams` method, the camera HAL can start requesting buffers again after the `configureStreams` call returns successfully. If the camera framework calls the `processCaptureRequest` method, the camera HAL can start requesting buffers during the `processCaptureRequest` call.

The semantics are different for the `signalStreamFlush` method and the `flush` method. When the `flush` method is called, the HAL can abort pending capture requests with `ERROR_REQUEST` to drain the pipeline as soon as possible. When the `signalStreamFlush` method is called, the HAL must finish all pending capture requests normally and return all buffers to the camera framework.

Another difference between the `signalStreamFlush` method and other methods is that `signalStreamFlush` is a one-way HIDL method, which means that the camera framework might call into other blocking APIs before the HAL receives the `signalStreamFlush` call. This means that

the `signalStreamFlush` method and other methods (specifically the `configureStreams` method) might arrive at the camera HAL in a different order than the order they were called in the camera framework. To address this asynchrony issue, the `streamConfigCounter` field was added to `StreamConfiguration` and added as an argument to the `signalStreamFlush` method. The camera HAL implementation should use the `streamConfigCounter` argument to determine whether a `signalStreamFlush` call arrives later than its corresponding `configureStreams` call.

You're diving into **Camera HAL3 Buffer Management APIs**, which are advanced but powerful improvements for how Android handles memory and buffer efficiency when working with cameras. Let me break down what you just read into **simpler concepts** and explain each piece clearly:

Purpose of These APIs

Android 10+ allows the **Camera HAL** to **request output buffers only when needed**, instead of receiving them along with every capture request.

This saves memory and improves latency.

Key Buffer Management APIs (Camera HAL ↔ Framework)

1. `requestStreamBuffers()`

- **Why:** HAL uses this to **ask for output buffers** from the framework.
- **When used:** Since HAL no longer gets buffers with the request, it must request them before the request reaches the final stages.
- **Can request multiple buffers** for multiple streams in one go (less IPC).
- **Use a high-priority thread** in HAL to avoid blocking pipeline performance.

Failure cases HAL must handle:

Reason	HAL Action
App disconnects stream	Send ERROR_REQUEST , continue normally
App holds buffers too long	Send ERROR_REQUEST , keep working
New stream config is happening	Wait for configureStreams to finish
HAL hits maxBuffers limit	Wait until some buffers are returned

2. `returnStreamBuffers()`

- **Why:** HAL uses this to **give back extra buffers** it requested but didn't use yet.
- **When needed:** If HAL has more buffers than active requests, it must release the unused ones.

- If HAL never over-requests, it doesn't need to call this.

3. 🚨 `signalStreamFlush()`

- **Why:** Framework tells HAL: "I'm going to reconfigure things. Return all buffers now."
- **What HAL should do:** Return all buffers — but **finish current requests cleanly**.
- **Don't abort requests** — unlike `flush()`, which can cancel them with `ERROR_REQUEST`.

Extra Note:

- This method is **one-way** (asynchronous). That means `signalStreamFlush()` might **arrive late** in the HAL, even after `configureStreams()` has arrived.
- To deal with this, Android added a `streamConfigCounter` to match flush calls with the right stream configuration session.

⚠ Flush vs. Signal Flush

Feature	<code>flush()</code>	<code>signalStreamFlush()</code>
Purpose	Abort ASAP and drain pipeline	Gently return all buffers for reconfig
Requests	Can be aborted with <code>ERROR_REQUEST</code>	Must be completed normally
Method type	Blocking (two-way)	Non-blocking (one-way)
Use case	Quick stop, e.g., app closes camera	Before reconfiguring streams

🏗 Implementation Checklist

To use buffer management in your HAL:

1. Implement `ICameraDevice@3.5`

2. Set characteristic:

cpp

```
android.info.supportedBufferManagementVersion = HIDL_DEVICE_3_5;
```

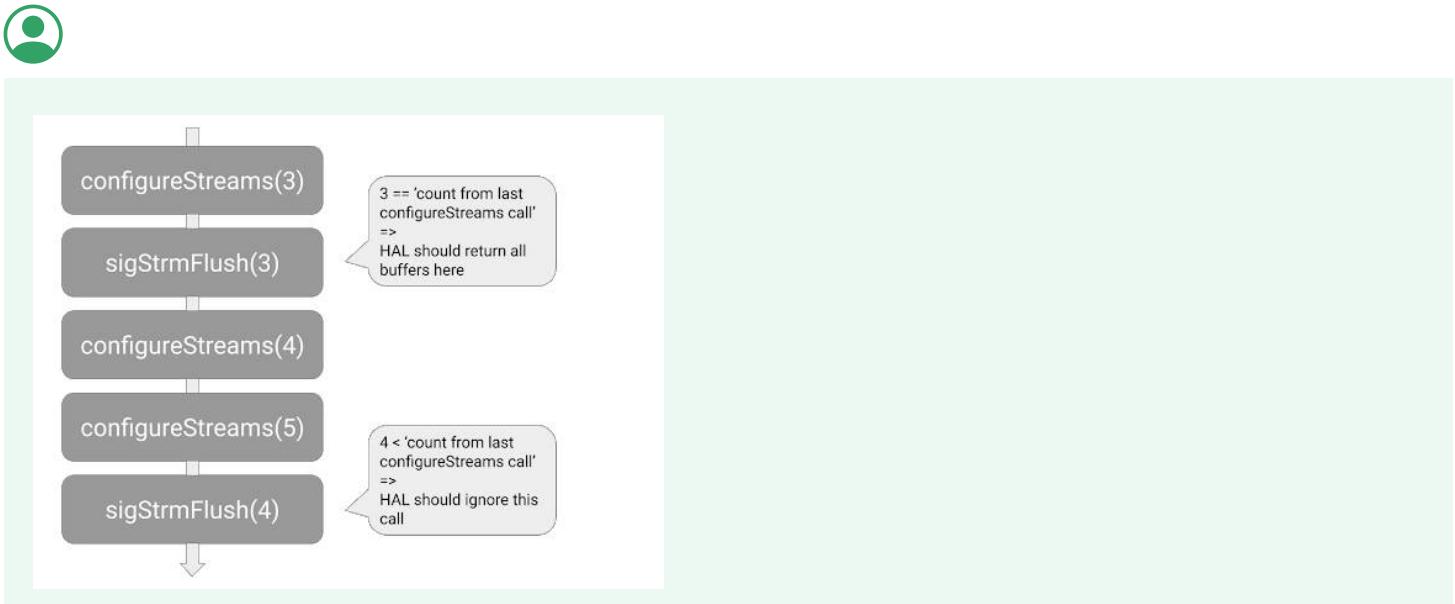
3. Implement these methods:

- `requestStreamBuffers()`
- `returnStreamBuffers()`
- `signalStreamFlush()`

✓ Summary – How it All Works Together:

- **Framework sends request** → contains **only control metadata** (no buffers).
- HAL queues it.
- **HAL requests buffer(s)** using `requestStreamBuffers()` before late stages.
- HAL produces results and calls `processCaptureResult()` to return results + buffers.
- If HAL has unused buffers → return them via `returnStreamBuffers()`.
- If framework needs to reconfigure streams → it calls `signalStreamFlush()`.

Let me know if you want a **code-style flow** or **sequence diagram** to tie this together!



This diagram explains how the Camera HAL handles `configureStreams()` and `signalStreamFlush()` (`sigStrmFlush`) calls, especially when they are received **out-of-order** or **late**.

📌 Context

In Android's Camera HAL interface, `configureStreams()` sets up new stream configurations (like size, format, usage). After each successful `configureStreams()`, the HAL increments a **stream configuration counter**.

`signalStreamFlush()` is used by the framework to tell the HAL to release all buffers for a specific stream configuration (usually for reconfiguration or cleanup).

🧠 Diagram Breakdown

Each function call is tagged with a number:

- `configureStreams(3)` means it's the third stream configuration.
- `sigStrmFlush(3)` is the flush call for that third config.
- This counter is **monotonically increasing** (increments every new config).

Let's analyze what's happening:

First Case:

- `configureStreams(3)`
- → `sigStrmFlush(3)`
 -  This flush **matches** the most recent config number.
 -  **HAL must flush buffers** for all active streams configured under (3).

Second Case:

- `configureStreams(4)`
- `configureStreams(5)`
- → `sigStrmFlush(4)`
 -  This flush is **for an older config** ($4 < 5$).
 -  **HAL must ignore** this call — it is stale and not relevant to the current configuration.

Rules Summary:

Condition	HAL Action
<code>flushConfigNumber == current</code>	Flush buffers
<code>flushConfigNumber < current</code>	Ignore the call (stale)
<code>flushConfigNumber > current</code>	Invalid (shouldn't happen)

When Is Memory Allocated?

Memory for **output buffers** (camera frames) is typically allocated:

- When `processCaptureRequest()` is called — **framework provides buffers**.
- **Before Android 10:** framework provided all N output buffers immediately for all requests in the pipeline.
- **From Android 10:** buffer provisioning is **decoupled** — HAL can request buffers **only when needed** using HAL3 buffer management APIs (`requestStreamBuffers()`), reducing memory use.

Metadata buffer is **not handled the same way**:

- It's usually passed as a part of the request or result structure (e.g., `CameraMetadata`).
- No explicit allocation/return APIs for metadata like there are for image buffers.

Would you like a step-by-step explanation of the full Android 10 buffer flow (`requestStreamBuffers()`, `returnStreamBuffers()`)?



Behavior changes when implementing the buffer management APIs

When using the buffer management APIs to implement the buffer management logic, consider the following possible behavior changes to the camera and camera HAL implementation:

Capture requests arrive at the camera HAL faster and more frequently: Without buffer management APIs, the camera framework requests output buffers for each capture request before sending a capture request to the camera HAL. When using the buffer management APIs, the camera framework no longer needs to wait for buffers and can therefore send capture requests to the camera HAL earlier.

Also, without buffer management APIs, the camera framework stops sending capture requests if one of the output streams of the capture request has reached the maximum number of buffers that the HAL can hold at one time (this value is designated by the camera HAL in the `HalStream::maxBuffers` field in the return value of a `configureStreams` call). With the buffer management APIs, this throttling behavior no longer exists and the camera HAL implementation must not accept `processCaptureRequest` calls when the HAL has too many capture requests queued.

`requestStreamBuffers` call latency varies significantly: There are many reasons a `requestStreamBuffers` call might take a longer time than average. For example:

For the first few buffers of a newly created stream, calls can take longer because the device needs to allocate memory.

The expected latency increases in proportion to the number of buffers requested in each call. The app is holding buffers and is busy processing. This can cause buffer requests to slow down or hit a timeout because of a lack of buffers or a busy CPU.

Buffer management strategies

The buffer management APIs allow for different kinds of buffer management strategies to be implemented. Some examples are:

Backward compatible: The HAL requests buffers for a capture request during the `processCaptureRequest` call. This strategy doesn't provide any memory savings, but can serve as the first implementation of the buffer management APIs, requiring very few code changes to the existing camera HAL.

Maximized memory savings: The camera HAL only requests output buffers immediately before

one is needed to be filled. This strategy allows for maximized memory savings. The potential downside is more camera pipeline jank when buffer requests take an unusually long time to finish.

Cached: The camera HAL caches a few buffers so that it's less likely to be affected by an occasional slow buffer request.

The camera HAL can adopt different strategies for particular use cases, for example, using the maximized memory saving strategy for use cases that use a lot of memory and using the backward-compatible strategy for other use cases.

Sample implementation in the external camera HAL

The external camera HAL was introduced in Android 9 and can be found in the source tree at hardware/interfaces/camera/device/3.5/. In Android 10, it has been updated to include ExternalCameraDeviceSession.cpp, an implementation of the buffer management API. This external camera HAL implements the maximized memory savings strategy mentioned in Buffer management strategies in a few hundred lines of C++ code.

Thanks! Let's break this down simply and clearly.

When does memory allocation happen for camera buffers?

It depends on how buffer management is implemented in the HAL:

Android Camera HALs can either:

1. Without buffer management APIs (old behavior):

- Framework allocates the output buffers **before** sending a capture request.
- HAL just receives a **buffer** already prepared.
- No manual allocation in HAL.
- **Throttling is done by framework:** If max buffers (`HalStream::maxBuffers`) are in use, it stops sending requests.
- Easier for HAL developer, less control.

2. With buffer management APIs (new behavior in Android 10+):

- HAL must call `requestStreamBuffers()` to get a buffer.
- **HAL is now responsible for memory management.**
- **Allocation happens lazily or strategically:**
 - First time HAL calls `requestStreamBuffers()` → allocation happens **here**.
 - Buffer reuse depends on strategy.

Common buffer request timing strategies:

Strategy	Description	Allocation Timing	Pros	Cons
Backward compatible	Request buffer inside <code>processCaptureRequest()</code>	During capture	Simple	No memory savings
Maximized memory savings	Request only just before using	Just-in-time	Low memory use	High risk of delay/jank
Cached	Request some buffers early and reuse them	Mixed	Fast response + memory savings	Complex logic

Key Behavior Changes When Using Buffer Management APIs

1. Requests come faster

- Framework doesn't wait for buffers.
- **HAL must throttle itself** (because framework won't anymore).
- You must keep track of how many active buffers you hold.

2. Buffer allocation can delay `requestStreamBuffers()`:

- First few requests = slow (due to memory allocation).
- Requesting many buffers at once = higher latency.
- App may hold buffers for long time = slowdown or timeout.

Real Example (from External Camera HAL)

In `ExternalCameraDeviceSession.cpp`:

- HAL uses `maximized memory savings` strategy.
- Calls `requestStreamBuffers()` **right before** filling a buffer.
- Only keeps buffers when needed.
- Tradeoff: Better memory use, but latency risk.

Summary

- **Memory allocation** for frame and metadata happens:
 - At the time of `requestStreamBuffers()` (if buffer is new).
 - Or earlier if using preallocation/caching.

- When using buffer management APIs:
 - Framework does not throttle → HAL must track buffer count.
 - Allocation delays can affect performance → use caching if needed.

Would you like a **diagram** or **sample buffer tracking logic** in code?



Session parameters

bookmark_border

The session parameters feature reduces delays by enabling camera clients to actively configure the subset of costly request parameters, that is, session parameters, as part of the capture session initialization phase. With this feature, your HAL implementations receive the client parameters during the stream configuration phase instead of the first capture request and can, depending on their values, prepare and build the internal pipeline more efficiently.

In Android 10, you can improve performance by using the optional session reconfiguration query feature for more control over the internal session parameter reconfiguration logic. For more information, see [Session reconfiguration query](#).

Examples and source

A reference session parameter implementation is already part of the CameraHal. This HAL uses the legacy Hal API. The binderized CameraHal that implements the camera HIDL API must use the respective HIDL sessionParams entry to access any new incoming session parameters during stream configuration.

Camera clients can query the keys of all supported session parameters by calling `getAvailableSessionKeys()` and eventually set their initial values through `setSessionParameters()`.

Implementation

Your CameraHal implementation must populate the `ANDROID_REQUEST_AVAILABLE_SESSION_KEYS` within the respective static camera metadata and provide a subset of `ANDROID_REQUEST_AVAILABLE_REQUEST_KEYS`, which contains a list of keys that are difficult to apply per-frame and can result in unexpected delays when modified during the capture session lifetime.

Typical examples include parameters that require a time-consuming hardware reconfiguration or an internal camera pipeline change. Control over session parameters can still be exerted in capture requests but clients should be aware of and expect delays in their app.

The framework monitors all incoming requests and if it detects a change in the value of a session parameter, it internally reconfigures the camera. The new stream configuration passed to CameraHal then includes the updated session parameter values, which are used to configure

the camera pipeline more efficiently.

Customization

You can define tags in the available session parameter list that is populated on the CameraHal side. This feature is not active if CameraHal leaves the available session parameter list empty.

Validation

CTS includes the following new cases for testing session parameters:

`CameraDeviceTest#testSessionConfiguration`

`CameraDeviceTest#testCreateSessionWithParameters`

`CameraDeviceTest#testSessionParametersStateLeak`

`NativeCameraDeviceTest#testCameraDevicePreviewWithSessionParameters`

In general, after a certain parameter is part of the session key list, its current value is included as part of the session parameters passed during stream configuration at the HAL layer.

Session parameters must be carefully selected. The values shouldn't change frequently, if at all, between stream configurations. Parameters that change frequently, such as capture intent, are ill-suited and adding them to the session parameter list could cause CTS failures due to excessive internal re-configuration.

Session reconfiguration query

Android 10 introduces an optional session reconfiguration query feature to improve performance as internal stream reconfigurations resulting from session parameter value modifications can reduce performance. To address this concern, HIDL `ICameraDeviceSession` version 3.5 and higher supports the `isReconfigurationRequired` method, which provides fine-grained control over the internal session parameter reconfiguration logic. Using this method, stream reconfiguration can occur precisely when required.

The arguments for `isReconfigurationRequired` provide the required information about every pending session parameter modification, allowing for various kinds of device-specific customizations.

This feature is implemented only in the camera service and the camera HAL. There are no public-facing APIs. If this feature is implemented, camera clients should see performance improvements when working with session parameters.

Implementation

To support session reconfiguration queries, you must implement the `isReconfigurationRequired` method to check whether complete stream reconfiguration is required for new session parameter values.

If the client changes the value of any advertised session parameter, the camera framework calls the `isReconfigurationRequired` method. Depending on the specific values, the HAL decides whether a complete stream reconfiguration is required. If the HAL returns false, the camera framework skips the internal reconfiguration. If the HAL returns true, the framework

reconfigures the streams and passes the new session parameter values accordingly.

The `isReconfigurationRequired` method can be called by the framework some time before a request with new parameters is submitted to the HAL, and the request can be cancelled before it is submitted. Therefore, the HAL must not use this method call to change its behavior in any way.

The HAL implementation must meet the following requirements:

The framework must be able to call the `isReconfigurationRequired` method at any time after active session configuration.

There must be no impact on the performance of pending camera requests. In particular, there must not be any glitches or delays during normal camera streaming.

The device and HAL implementation must meet the following performance requirements:

Hardware and software camera settings must not be changed.

There must be no user-visible impact on camera performance.

The `isReconfigurationRequired` method takes the following arguments:

`oldSessionParams`: Session parameters from the previous session. Usually the existing session parameters.

`newSessionParams`: New session parameters that are set by the client.

The expected return status codes are:

`OK`: Successful reconfiguration required query.

`METHOD_NOT_SUPPORTED`: The camera device doesn't support the reconfiguration query.

`INTERNAL_ERROR`: The reconfiguration query can't complete due to an internal error.

The return values are:

`true`: Stream reconfiguration is required.

`false`: Stream reconfiguration isn't required.

To ignore a session reconfiguration query, the HAL returns `METHOD_NOT_SUPPORTED` or `false`.

This results in the default camera service behavior where stream reconfiguration is triggered on each session parameter change.

Validation

The session reconfiguration query feature can be validated using the VTS test case in `CameraHidlTest#configureStreamsWithSessionParameters`.

Let's break this complex **session parameter** and **session reconfiguration** concept into simple parts to help you understand it clearly.

🔍 What Are Session Parameters?

Session parameters are **special camera parameters** that:

- Are **expensive to change** (e.g., need internal reconfiguration or hardware switch).
 - Don't change often during a session.
 - Should be **set once** during stream configuration (not per-frame like most request keys).
-

Why They Exist

Without this feature:

- The first time a slow-to-change parameter arrives in a capture request, the HAL may **stall**, causing **jank or latency**.

With session parameters:

- Camera client sets those costly parameters early during session creation.
 - HAL prepares everything during **stream configuration**.
 - Result: **Smoother performance, less runtime reconfiguration**.
-

How It Works

◆ During Session Creation

1. **Framework** checks which parameters are session parameters.
 2. It sends them to HAL in the `configureStreams()` call.
 3. HAL configures the camera pipeline based on these.
-

◆ If Parameters Change Later

- If a session parameter value changes (e.g., ISO mode requiring hardware switch):
 - The framework calls `isReconfigurationRequired()`.
 - HAL compares old vs new values:
 - If **reconfiguration needed**, return `true`.
 - If **not needed**, return `false`.
-

What Must HAL Do?

Minimum:

- Implement `ICameraDeviceSession@3.5`
- Populate:
 - `ANDROID_REQUEST_AVAILABLE_SESSION_KEYS`
 - Subset of `ANDROID_REQUEST_AVAILABLE_REQUEST_KEYS` (expensive-to-change keys)

Implement:

- `isReconfigurationRequired(oldSessionParams, newSessionParams)`
 - Must not alter HAL behavior.
 - Must return result quickly.
 - Must avoid any camera glitch or delay.

Behavior Overview

Stage	With Session Parameters
<code>configureStreams()</code>	Includes session parameter values
<code>processCaptureRequest()</code>	No need to set costly parameters repeatedly
Parameter value changes	Framework asks HAL: <code>isReconfigurationRequired()</code>
HAL says <code>true</code>	Framework triggers full stream reconfiguration
HAL says <code>false</code> or <code>METHOD_NOT_SUPPORTED</code>	Framework skips reconfiguration

Validation

CTS Tests:

- `CameraDeviceTest#testCreateSessionWithParameters`
- `CameraDeviceTest#testSessionParametersStateLeak`

VTS Test:

- `CameraHidlTest#configureStreamsWithSessionParameters`

Example: Good Session Parameters

Parameter	Good for Session Parameter?	Why?
Noise Reduction Mode	<input checked="" type="checkbox"/> Yes	May require ISP reconfiguration
Edge Enhancement Mode	<input checked="" type="checkbox"/> Yes	Impacts pipeline behavior

Parameter	Good for Session Parameter?	Why?
Control Mode	✗ No	Changes often per frame
Capture Intent	✗ No	Frequently varies

🎯 Key Benefits

- Reduces frame delay and camera "jank"
- Enables smoother session start-up
- Avoids costly mid-session hardware reconfigurations

Would you like a **code template** for how `isReconfigurationRequired()` might look in a real HAL implementation?



Why do we need this? explain in simple words

Great question. Let's explain **why we need session parameters in very simple words**:

📸 Imagine you are using a camera app...

- You open the app.
- You start recording a video.
- Suddenly, the app **changes a setting** like "noise reduction" or "high-quality mode".

🐢 What happens without session parameters?

- That change needs the **hardware to reconfigure** (like rewiring the camera chip).
- The reconfiguration takes time.
- So your app **lags**, the preview **freezes**, or a frame is **missed**.

✓ What happens with session parameters?

- The app tells the camera HAL **in advance**:

"Hey, I'll be using high-quality noise reduction."
- The HAL **prepares everything** when the session starts.

- Now during recording, nothing needs to be reconfigured — it just works smoothly.

Simple Analogy:

Imagine you're **making tea**:

- Without session parameters: You keep changing sugar and milk amounts **after the tea is made**. You have to throw it away and remake it each time. Waste of time.
- With session parameters: You tell the cook **in advance** how you like your tea. It's made right the first time — no need to redo.

So, in one sentence:

Session parameters help the camera know expensive settings ahead of time, so it can prepare early and avoid slowdowns or glitches later.

Would you like a real-world example using something like "night mode" or "beauty mode"?



yes

Perfect! Let's take a **real-world example** using **night mode** and **beauty mode** on a camera:

Example 1: Night Mode (Low-light Photography)

With Session Parameters:

- The app says **before starting the camera**:
"I'll be using night mode — that means high noise reduction, long exposure, and low frame rate."
- HAL prepares the camera hardware accordingly.
- When you start capturing, everything works **smoothly** — no hiccups, no lag, and night mode looks great.

Without Session Parameters:

- App starts the session normally.
- Later, it suddenly says:

"Oops! I need night mode now."

- HAL needs to:
 - Stop preview
 - Reconfigure camera hardware (slow)
 - Restart stream
 - You get a **freeze**, **frame drop**, or **lag** — user thinks camera is broken.
-

Example 2: Beauty Mode (Face Smoothing)

With Session Parameters:

- App tells the HAL at start:

"I'll apply beauty filters like skin smoothing and eye enhancement."
- HAL configures image processing pipeline accordingly (adds shaders or adjusts ISP blocks).

Without Session Parameters:

- Beauty filter is applied halfway through.
- HAL is caught off guard:
 - Needs to rebuild internal processing graph
 - Stream might pause briefly
 - Frames might be skipped

So What's the Point?

Apps **avoid sudden reconfigurations** during a session by giving HAL a **heads-up** on expensive settings. That's what **session parameters** are.

Let me know if you'd like this shown with a diagram or camera capture flow!



Single producer, multiple consumer camera buffer transport

bookmark_border

This feature introduces a set of methods that allows camera clients to add and remove output surfaces dynamically while the capture session is active and camera streaming is ongoing. A

new output can map to a specific, user-selected shared camera stream. After a surface is added, it can be removed at any time.

The general idea is to share the buffers associated with a particular camera stream within several output surfaces. An internal reference counter keeps track of the buffers as they become ready for further processing on the consumer side. When all consumers complete their respective tasks the buffer gets dequeued and is available for the camera.

You're referring to the "**Single Producer, Multiple Consumer (SPMC)**" model in the Android Camera HAL buffer transport system — particularly relevant for **shared camera streams**. Let me explain it in **simple terms** with visuals and real-world analogies.

What Is Happening?

- **Producer:** The **camera HAL** (writes the image data into buffers).
- **Consumers:** Multiple **surfaces** or **apps** that read/process/display this image (e.g., preview UI, face detection, video encoder).
- The **same frame** needs to be sent to multiple places **at the same time**.

Real-World Analogy: Security Camera System

Imagine:

- A security camera is **capturing footage** (Producer).
- The same live feed goes to:
 - The guard's monitor (Preview).
 - A recording DVR (Video encoder).
 - An AI system checking faces (ML client).

The camera **doesn't duplicate the video stream** three times. Instead, it shares the same video frame with all 3 outputs.

This is exactly how **Single Producer, Multiple Consumer** works.

Why Is This Needed?

Without SPMC:

- HAL would need to **copy each frame multiple times**.
- Wastes **memory** and **performance**.

- Latency increases.
- Power consumption goes up.

With SPMC:

- One buffer is produced.
- Multiple consumers get access to it via shared handles (e.g., `AHardwareBuffer` OR `GraphicBuffer`).
- Consumers **signal when they're done**.
- Only when **all** consumers are done, the buffer is returned to HAL for reuse.

🔧 Key Concepts

Component	Role
Producer (HAL)	Writes image data into buffer.
Consumers (Apps)	Read/process the buffer (e.g., encode, display, ML).
Reference Counter	Tracks how many consumers are still using the buffer.
Surface	Destination that receives frames (can be added/removed dynamically).
Dequeue	Buffer is free and reused only after all consumers finish .

📈 Flow Diagram (Simplified)



🧪 Dynamic Add/Remove Surfaces

You can **add/remove surfaces at runtime**:

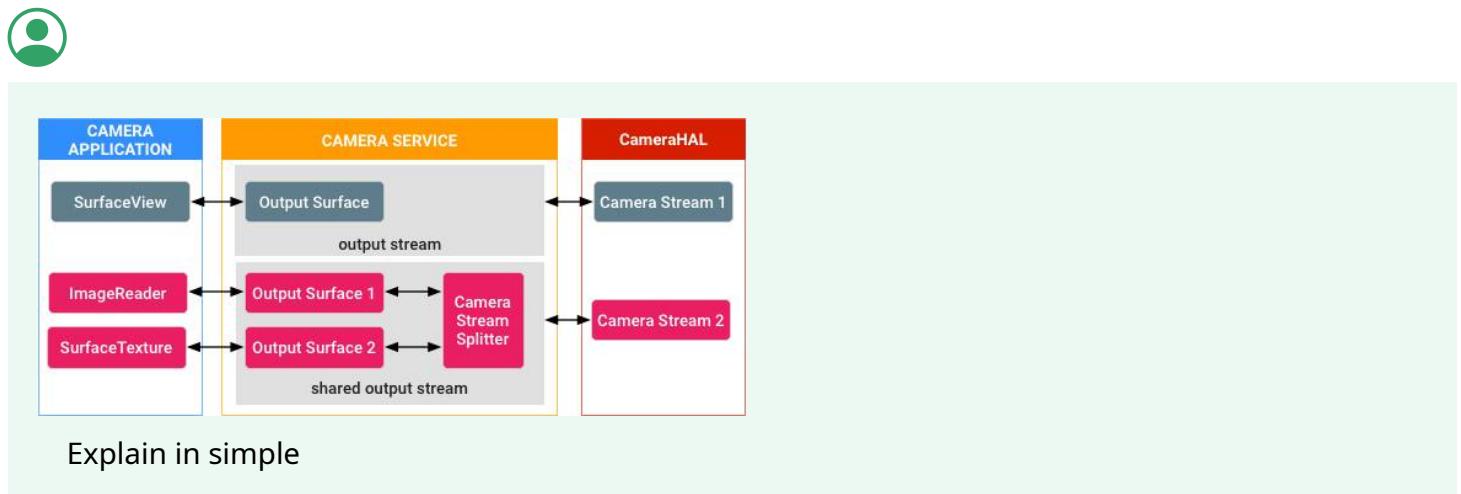
- Start with just preview.
- Add ML surface later when user enables "Smart Capture".
- Remove encoder when video recording stops.

The HAL doesn't need to reconfigure the camera — it just updates who receives frames.

Summary

Feature	Benefit
Single Producer, Multiple Consumer	One buffer shared, zero copy overhead
Reference-counted buffer reuse	Ensures data isn't lost or reused prematurely
Dynamic surface add/remove	Enables flexible camera use-cases
Efficient for battery + memory	Great for mobile constraints

Let me know if you'd like this explained with **source-level AOSP details**, like how **Camera3Stream**, **Surface**, or **GraphicBuffer** works internally.



This diagram shows **how the Android camera system can send the same camera data to multiple places at once** — like preview, processing, and recording — using **shared buffers**.

Simple Explanation:

Imagine your camera is like a faucet, and you want to send the water (camera frames) to multiple buckets (apps or features).

There are **two types of camera output streams** here:

◆ 1. Single Output Stream (Top part)

- The camera data goes from **CameraHAL** → to one **Output Surface** → to **SurfaceView** in your app.
 - This is **one-to-one** — the frame goes to one place only.
 - Good for simple preview.
-

◆ 2. Shared Output Stream (Bottom part)

- The camera stream is duplicated using a **Camera Stream Splitter** inside the **Camera Service**.
- It sends the same frame to **multiple surfaces**:
 - **ImageReader** (for saving a photo)
 - **SurfaceTexture** (for applying filters or preview)

So:

- **CameraHAL** sends frame → to **Camera Stream 2** → Camera Service **splits it** → sends it to both Output Surface 1 and 2.
 - Both consumers process the frame in their own way.
-

What happens to the buffer?

- There's **a reference counter**.
- Camera waits until **both consumers finish using** the buffer.
- Then the buffer is returned (dequeued) back to the camera.

Why this is useful?

- You don't need to duplicate work in the HAL.
- Camera can **stream to multiple destinations efficiently**.
- Helps in use cases like:
 - Showing preview
 - Recording video
 - Running ML models in parallel
 - Taking snapshots

Let me know if you'd like a real-world analogy too!



Examples and source

The core implementation of this feature can be found in the Camera3StreamSplitter module. Documentation on this feature can be found in the developer reference:

`updateOutputConfiguration()`

`addSurface()`

`removeSurface()`

Implementation

No implementation is required on the Camera HAL side as this feature is implemented on the framework side.

Validation

Your implementation must pass CTS cases that cover this feature from the MultiViewTest module and the native JNI library for the native API.

You're referring to the **shared output stream support in Android Camera Framework**, where multiple app surfaces can receive frames from the same camera stream using a **software-based buffer splitter**.

Let's go over each point:

✓ What is Camera3StreamSplitter?

`Camera3StreamSplitter` is a **framework-side software module** in Android's Camera stack that allows a **single stream of camera frames** to be duplicated and sent to **multiple consumers** (surfaces) like:

- `ImageReader` (for JPEG capture)
- `SurfaceTexture` (for OpenGL preview)
- `MediaRecorder` (for video recording)

This avoids duplicating stream effort on the HAL side.

🔧 Core Location in AOSP

📁 AOSP Source File:

cpp

frameworks/av/services/camera/libcameraservice/device3/Camera3StreamSplitter.cpp

📄 Header:

cpp

frameworks/av/services/camera/libcameraservice/device3/Camera3StreamSplitter.h

Related APIs in Camera2 Java API

Method	Purpose
<code>updateOutputConfiguration()</code>	Updates an existing output stream (e.g., adding/removing surfaces).
<code>addSurface()</code>	Adds a new surface to the output configuration for splitting.
<code>removeSurface()</code>	Removes a surface from the stream.

These APIs let your app dynamically control which surfaces receive the shared frames.

Example Use Case

Imagine this Java-side scenario:

```
java

OutputConfiguration outputConfig = new OutputConfiguration(surface1);
outputConfig.addSurface(surface2); // Share the same camera stream

sessionConfiguration.addOutputConfiguration(outputConfig);
cameraDevice.createCaptureSession(sessionConfiguration, callback, handler);
```

 Internally, the **framework uses Camera3StreamSplitter** to copy the frames and send them to both **surface1** and **surface2**.

Validation: CTS and MultiViewTest

To test and verify this feature:

- CTS (Compatibility Test Suite) has:

```
swift

cts/tests/tests/hardware/src/android/hardware/camera2/cts/MultiViewTest.java
```

- It tests shared output streaming with multiple **SurfaceView**, **ImageReader**, etc.
- JNI-level native validation also exists for NDK/Native camera use.

! HAL Requirements

"No implementation is required on the Camera HAL side."

Yes — the HAL is unaware of the splitting. From HAL's view, it's a **single stream**. The buffer duplication and forwarding is **purely in the framework (Camera Service)**.

Summary

Feature	Details
 Split frames	Camera3StreamSplitter
 Location	frameworks/av/services/camera/libcameraservice/device3/
 Java APIs	addSurface(), removeSurface(), updateOutputConfiguration()
 HAL Work	Not needed
 Tests	MultiViewTest in CTS

Would you like:

- A full working sample with Java + NDK camera surfaces?
- A deep dive into **Camera3StreamSplitter.cpp** logic (like buffer queueing, fence handling)?



stream vs session

Great question! In the Android **Camera HAL and framework**, the terms **stream** and **session** refer to **different levels of abstraction** in the image capture pipeline. Here's a simple breakdown:

Camera Session (High-level concept)

- **What it is:**
A **camera capture session** is a group of **output targets** (like **SurfaceView**, **ImageReader**, etc.) that you can send requests to.
- **Framework Class:**
Java: [CameraCaptureSession](#)
Native: [ACameraCaptureSession](#)
- **What it manages:**
 - Group of output surfaces
 - One or more capture requests
 - Controls how images are captured and sent to outputs

- **Analogy:**

Think of a *session* like a “photo shoot setup.”

You define:

- What camera settings to use (focus, exposure)
- Where the pictures should go (screen, file, etc.)

Camera Stream (Lower-level concept inside a session)

- **What it is:**

A **stream** is a continuous flow of image data going from the camera device to a single or multiple surfaces (like `SurfaceView` or `ImageReader`).

- **What it represents:**

- One actual image pipeline (e.g., a buffer queue)
- Has specific resolution, format, and buffer usage
- Internally backed by a `GraphicBuffer` queue

- **Created by:**

`Camera3Device` in the framework, or by the app using `OutputConfiguration`

- **Example:**

If you’re previewing and recording video at the same time, you may have:

- Stream 1 → for preview (`SurfaceView`)
- Stream 2 → for video recording (`MediaRecorder`)
- Stream 3 → for JPEG stills (`ImageReader`)

Relationship

Concept	Description
Session	Owns and manages multiple streams
Stream	Sends data to one or more surfaces
CaptureRequest	Tells which stream(s) to use during a capture
Surfaces	The actual targets — passed in via <code>OutputConfiguration</code>

Real-world Example

In Java:

```
java
```

```
List<Surface> surfaces = Arrays.asList(surfaceView, imageReaderSurface);
cameraDevice.createCaptureSession(surfaces, callback, handler);
```

- This creates **one session**
- The session has **two output streams** (preview + still capture)
- You can send **CaptureRequests** to either or both streams

In the HAL:

- `camera3_capture_request_t` refers to **streams** by their stream IDs.
- The HAL sees streams as buffer producers/consumers.
- The framework builds a session by configuring these streams together.

Summary Table

Feature	Stream	Session
Definition	A flow of camera image buffers	A group of configured outputs
Represents	One output pipeline	The whole capture environment
Granularity	Low-level (buffer level)	High-level (capture control)
Count	Many streams per session	One session active at a time
Used by	HAL	App / Framework

Let me know if you'd like:

- A diagram showing session ↔ stream ↔ surface
- NDK vs Java API comparison for these terms
- Internal call trace (`Camera3Device` → `Camera3OutputStream` VS `Camera3CaptureSession`)



Do requests sent out for every frame?

Yes — **a request is typically sent for every frame**, but with some flexibility. Let's break it down simply:

What is a CaptureRequest?

A **CaptureRequest** tells the camera **how to capture one frame**:

- Exposure, white balance, focus, crop, etc.
- Which **streams/surfaces** to send image data to

⌚ Are requests sent every frame?

YES — one request per frame (most of the time):

- For **continuous preview** (like `Camera2` preview stream):

A repeating request is sent using:

```
java
session.setRepeatingRequest(captureRequest, callback, handler);
```

This repeats **the same request** for each frame, at 30/60fps, etc.

- For **manual control**, you can:

```
java
session.capture(singleFrameRequest, callback, handler);
```

→ This sends a **new request for a single frame** (e.g. for a photo).

⌚ Flow: Repeating Requests = Auto Frame Loop

```
plaintext
App → setRepeatingRequest(req)
    ↳ Framework schedules: Frame 1 → req
        Frame 2 → req
        Frame 3 → req
        ...
        ...
```

So yes, the **same or different requests** are sent for each frame, depending on what you configure.

🧠 What about in HAL?

At the HAL level:

- The **framework sends one `camera3_capture_request_t` per frame**.
- HAL must process it and return the result (`camera3_capture_result_t`) for **that frame**.
- Requests have **frame numbers** (incrementing counters).

So even if the app repeats the same settings, the framework still wraps them in a new request each time.

Example (30 fps preview):

Frame #	Request Sent?	Request Contents
1	<input checked="" type="checkbox"/>	Focus = Auto, Stream = Preview
2	<input checked="" type="checkbox"/>	Same (repeating request)
3	<input checked="" type="checkbox"/>	Same
...	<input checked="" type="checkbox"/>	...

Summary

Question	Answer
Is a request sent per frame?	<input checked="" type="checkbox"/> Yes
Can it be the same request each time?	<input checked="" type="checkbox"/> Yes (via repeatingRequest)
Can each frame have different settings?	<input checked="" type="checkbox"/> Yes (send new CaptureRequest each frame)
Is this visible to the HAL?	<input checked="" type="checkbox"/> Yes, HAL receives one request per frame

Let me know if you want:

- The internal **Camera3Device** → HAL request flow
- How request/result are matched by frame number
- Timeline view for request–buffer–result flow



10-bit camera output

bookmark_border

For devices running Android 13 and higher, Android supports 10-bit camera output through dynamic range profiles that can be configured by the camera client as part of the stream configuration. Device manufacturers can add support for 10-bit dynamic range profiles such as HLG10, HDR 10, HDR 10+, and Dolby Vision.

10-bit camera output support lets camera clients discover supported 10-bit dynamic range profiles of a device by calling `getSupportedProfiles`. The framework then returns an instance of `DynamicRangeProfiles`, which includes information about supported dynamic range profiles and, if available, capture request constraints. The HLG10 profile must be supported. The recommended dynamic range profile is listed in the

REQUEST_RECOMMENDED_TEN_BIT_DYNAMIC_RANGE_PROFILE field.

Camera clients can configure stream combinations by calling setDynamicRangeProfile. For more information on mandatory output stream combinations, see the 10-bit output additional guaranteed configurations table in Regular capture.

Requirements

To support 10-bit camera output, the device must have a 10-bit or higher capable camera sensor with respective ISP support. For details about related compatibility requirements for 10-bit support, see section 7.5. Cameras in the CDD.

Implementation

To provide support for 10-bit camera output, device manufacturers must perform the following Camera AIDL HAL integrations:

Include ANDROID_REQUEST_AVAILABLE_CAPABILITIES_DYNAMIC_RANGE_TEN_BIT in camera capabilities.

Populate ANDROID_REQUEST_AVAILABLE_DYNAMIC_RANGE_PROFILES_MAP with all supported dynamic range profiles and a bitmap of their constraints. The HLG10 profile must be supported. You must also include a recommended dynamic range profile to inform camera clients of the optimal supported format.

Ensure support for the dynamic range profile value during stream configuration for streams using the P010 format or support for an implementation-defined format (ImageFormat.PRIVATE).

Depending on the dynamic range profile, set the static or dynamic metadata buffer of processed Gralloc 4 buffers before notifying the camera service.

For further details on 10-bit camera output in the Camera HAL, see the following in metadata_definitions.xml:

DYNAMIC_RANGE_TEN_BIT

HAL details for availableDynamicRangeProfilesMap

recommendedTenBitDynamicRangeProfile

10BIT_OUTPUT

For a reference Camera HAL implementation supporting 10-bit camera output, see /hardware/google/camera/devices/EmulatedCamera/hwl.

Validation

To validate your implementation of 10-bit camera output and ensure that third-party apps can enable the feature, we recommend performing the following three stages of validation.

Test API functional correctness

Compare native camera and third-party app

Compare standard dynamic range and high dynamic range

For visual validation of 10-bit camera output, it's assumed that the device supports displaying HDR (1000+ nits display), and the video viewing app (for example, Google Photos) supports playback of HDR video.

Test API functional correctness

To test the API functional correctness of 10-bit camera output, run the following CTS, camera ITS, and VTS tests:

hardware/interfaces/camera/provider/aidl/vts/: Tests for basic discovery, configuration, and streaming, and checks for the presence of HDR metadata where required.

tests/camera/src/android/hardware/camera2/cts/: Ensures that the camera behaves according to the AOSP API specifications.

cts/apps/CameraITS: Confirms general video behavior is consistent when HDR profiles are used. The specific test is tests/scene4/test_video_aspect_ratio_and_crop.py.

Compare native camera and third-party app

We strongly recommend ensuring that the results of capturing 10-bit videos with a third-party app are similar, if not identical, to the native camera app. This means that tuning choices, such as exposure, dynamic range, and color, should carry forward from the native app to third-party apps. To verify the video recording behavior of a third-party app supporting 10-bit camera output on your device, use the Camera2Video sample app on GitHub. The following guidance serves to illustrate the visible aspects of HDR without objective numbers, due to the variability of sensors, panels, viewing conditions, and vendor preferences.

Suggested scenes for comparison

To make a comparison between the native camera app and a third-party app, capture videos using several different scenes with both the native camera app and the Camera2Video sample app. The following are suggested scenes to use for comparison:

A mid-light to low-light scene with a bright object, such as a candle or small bright light that creates a significant range of brightness. This confirms the auto exposure behavior and dynamic range.

A bright outdoor scene with vibrant colors and reflective objects such as chrome bumpers on a car, which creates bright highlights. This confirms the rendering for bright scenes with even brighter highlights.

A mid-range, low dynamic range scene such as an indoor natural scene in a home or office. This confirms that less extreme lighting conditions behave as expected.

For all scenes, we recommend having people and faces to verify exposure, color, and skin tone handling. Reducing shot-to-shot variation eases back-to-back comparisons.

Compare standard dynamic range and high dynamic range

To ensure that there's a perceived benefit of using a 10-bit dynamic range profile over a standard dynamic range profile, compare video captures using SDR (no HDR profile) against HDR videos to confirm that key aspects of HDR appear in the captures. To compare SDR and HDR, use the Camera2Video sample app and suggested scenes for comparing the native camera app and third-party apps.

The following are key aspects to verify in the suggested scenes. Display panels capable of HDR vary in brightness levels (measured in nits or lumens), so the following numbers given are meant to be examples:

In the mid-light to low-light scene, the bright highlights of the candle or small light are rendered at max brightness for the display (possibly up to 1000 nits) in the HDR clip, and rendered at max brightness for SDR (approximately 100 nits) in the SDR clip. In the HDR clip, the bright highlights should shine out of the display, capturing the user's perception of what the scene's true dynamic range was. Compared to the HDR clip, the SDR clip should appear as flatter and less bright.

In the bright output scene, depending on the device's tuning, the HDR clip shows an apparent difference in screen brightness as compared to the SDR clip. For the HDR clip, the screen brightness for the overall scene (depending on headroom) should be higher, for example, up to 800 nits, and even more so for the bright highlights such as the chrome bumpers, around maximum brightness.

In the mid-range, low dynamic range indoor capture, the HDR and SDR clips are similar in color and tone, with the HDR capture being potentially brighter than the SDR. The HDR shouldn't be darker than the SDR. If tuning choices make this impossible, ensure that the third-party app behavior matches the native camera app behavior.

Here's a **simple explanation** of the **10-bit camera output** feature in Android 13+:

What is 10-bit camera output?

It means the camera can capture **more color and brightness detail** in each frame — **10 bits per color channel** instead of the standard 8 bits.

- **8-bit = 256 levels per color**
- **10-bit = 1024 levels per color**
- This allows for better **HDR (High Dynamic Range)** video and photos — with **brighter highlights, deeper shadows, and richer colors**.

Why do we need this?

Because modern displays (especially HDR ones) can **show more brightness and color detail**, and the camera should match that capability.

This helps:

- Videos look more **realistic and vibrant**
- Bright lights (like sunlight or candles) don't get "blown out"
- Dark scenes preserve **shadow detail**

How does Android support it?

Starting in **Android 13**, camera apps can:

1. **Check** if the device supports 10-bit modes like:
 - **HLG10, HDR10, HDR10+, Dolby Vision**
2. **Set** the desired HDR profile (via `setDynamicRangeProfile`)
3. **Capture** in higher-quality 10-bit format (like `P010`)

The camera service and HAL **coordinate this** using new metadata and stream configurations.

What must the device support?

- A **camera sensor** and **ISP** that can actually produce 10-bit images
- HAL must advertise:
 - `ANDROID_REQUEST_AVAILABLE_CAPABILITIES_DYNAMIC_RANGE_TEN_BIT`
 - Supported profiles in `ANDROID_REQUEST_AVAILABLE_DYNAMIC_RANGE_PROFILES_MAP`
- HAL must support 10-bit formats (e.g., `P010` or `PRIVATE`)

How is it tested?

- With **CTS/VTS/ITS tests** to check:
 - API behavior
 - Video quality
 - Correct metadata
- Visual validation:
 - Compare 10-bit video vs 8-bit
 - Use real scenes (candlelight, chrome in sunlight, indoor colors)
 - Use both native and 3rd-party camera apps to check consistency

TL;DR

Feature	What it means
10-bit output	More precise colors and brightness (HDR support)
Dynamic range profiles	Tells camera how to output video (HLG10, HDR10, etc.)
Used for	HDR video, better color grading, higher-end recording
Requires	Supported sensor, ISP, and HAL metadata
Verified by	CTS/VTS/ITS tests + visual comparison

Let me know if you want an example flow, code snippet, or comparison chart for SDR vs HDR!



Camera bokeh

bookmark_border

Camera bokeh is a shallow depth-of-field effect created by blurring out parts of a scene so that they're not in focus. On cameras on mobile devices, bokeh is achieved using depth information acquired from the stereo vision from two cameras or from dual photodiodes (PDs) of a single camera.

Starting from Android 11, the Android platform supports bokeh implementations and provides APIs to make the feature available to third-party apps.

Implementation

To make the camera bokeh feature available on your device, do the following:

Advertise ANDROID_CONTROL_AVAILABLE_EXTENDED_SCENE_MODE_MAX_SIZES:

This tag is in the format of a three-integer tuple array, with each tuple in the format of {mode, maxWidth, maxHeight}. In addition to

{ANDROID_CONTROL_EXTENDED_SCENE_MODE_DISABLED, 0, 0}, the camera HAL must also list one or both of the ANDROID_CONTROL_EXTENDED_SCENE_MODE_BOKEH_STILL_CAPTURE and ANDROID_CONTROL_EXTENDED_SCENE_MODE_BOKEH_CONTINUOUS modes with their corresponding maximum streaming sizes.

Advertise ANDROID_CONTROL_AVAILABLE_EXTENDED_SCENE_MODE_ZOOM_RATIO_RANGES:

This tag is in the format of a {minZoomRatio, maxZoomRatio} array, which contains all the zoom ratio ranges for an enabled extended scene mode, in the same order as in ANDROID_CONTROL_AVAILABLE_EXTENDED_SCENE_MODE_MAX_SIZES. A zoom range of [1.0, 1.0] means that zoom isn't supported.

Populate ANDROID_CONTROL_USE_EXTENDED_SCENE_MODE in ANDROID_CONTROL_AVAILABLE_MODES.

For an app to trigger the bokeh feature, the app must set ANDROID_CONTROL_MODE to ANDROID_CONTROL_USE_EXTENDED_SCENE_MODE and ANDROID_CONTROL_EXTENDED_SCENE_MODE to one of supported extended scene modes. Note that implementing bokeh might lead to extra memory consumption because of stereo computation.

If a supported extended scene mode can't be applied per frame and results in unexpected delays when enabled/disabled, include ANDROID_CONTROL_EXTENDED_SCENE_MODE in

ANDROID_REQUEST_AVAILABLE_SESSION_KEYS. At the same time, implement the ICameraDeviceSession::isReconfigurationRequired() method to avoid reconfiguration for extended scene modes that don't require reconfiguration.

Validation

To validate the bokeh feature on your device, run the following CTS, VTS, and CTS Verifier tests:

CtsCameraTestCases

VtsHalCameraProviderV2_4TargetTest

CameraBokehTest in CTS Verifier

Here's a simple explanation of **Camera Bokeh** in Android:

What is Camera Bokeh?

Bokeh is the blurry background effect you see in professional photos — it makes the subject stand out sharply while the background is out of focus.

On phones, this is **simulated** using:

- Two cameras (stereo vision) to detect depth
- Or a single camera with **dual photodiodes** (PDs) that sense depth by comparing slight differences in light.

Android Support

Since **Android 11**, Android has **built-in support for bokeh**, so third-party apps (like Instagram, Snapchat) can use it too — if your hardware and HAL support it.

How to Implement (HAL & Framework-side):

1. Advertise supported bokeh modes

- Using **ANDROID_CONTROL_AVAILABLE_EXTENDED_SCENE_MODE_MAX_SIZES**
 - You declare things like:

```
plaintext

{BOKEH_STILL_CAPTURE, 1920, 1080}
{BOKEH_CONTINUOUS, 1280, 720}
```

- Also include:

```
plaintext
{EXTENDED_SCENE_MODE_DISABLED, 0, 0}
```

2. Advertise supported zoom range

- Using `ANDROID_CONTROL_AVAILABLE_EXTENDED_SCENE_MODE_ZOOM_RATIO_RANGES`
 - Example: `[1.0, 2.0]` (zoom supported), `[1.0, 1.0]` (no zoom)

3. Enable apps to use the mode

- In your capture request:

```
plaintext
ANDROID_CONTROL_MODE = USE_EXTENDED_SCENE_MODE
ANDROID_CONTROL_EXTENDED_SCENE_MODE = BOKEH_CONTINUOUS or BOKEH_STILL_CAPTURE
```

4. Handle reconfiguration if needed

- If enabling bokeh needs reconfig (more memory, latency), implement:
 - `ICameraDeviceSession::isReconfigurationRequired()`
 - Also add `ANDROID_CONTROL_EXTENDED_SCENE_MODE` to `SESSION_KEYS`

✓ Validation

Run the following tests to ensure it's working:

Test Suite	Purpose
<code>CtsCameraTestCases</code>	Tests camera bokeh behavior at app level
<code>VtsHalCameraProviderV2_4TargetTest</code>	Verifies HAL-side compliance
<code>CameraBokehTest</code> (CTS Verifier)	Manual test to verify real-world behavior

🎯 Summary

- **Why:** To simulate professional shallow-depth photography
- **How:** Use stereo/PD hardware, HAL tags, and framework API
- **Apps:** Can trigger it using the standard Android Camera2 API
- **Extra:** Needs more memory and sometimes reconfiguration

Would you like a code/config snippet example or HAL integration example for this?



Concurrent camera streaming

bookmark_border

Android allows devices to support concurrent streaming of camera devices. For example, this allows a device to have both the front and back cameras operating at the same time. From Android 11, the Camera2 API includes the following methods that apps can call to determine if the cameras support concurrent streaming and the stream configurations that are supported.

`getConcurrentCameraIds`: Gets the set of combinations of currently connected camera device identifiers that support configuring camera device sessions concurrently.

`isConcurrentSessionConfigurationSupported`: Checks whether the provided set of camera devices and their corresponding session configurations can be configured concurrently.

A set of mandatory stream combinations that must be supported during concurrent streaming are included through a camera device's camera characteristics in the `SCALER_MANDATORY_CONCURRENT_STREAM_COMBINATIONS` property.

Each camera device advertised through `getConcurrentStreamingCameraIds()` must support the following guaranteed configurations for concurrent streams.

Target 1 Target 2

Type	Max size	Type	Max size	Sample use cases
------	----------	------	----------	------------------

YUV	s1440p			In-app video or image processing
-----	--------	--	--	----------------------------------

PRIV	s1440p			In-app viewfinder analysis
------	--------	--	--	----------------------------

JPEG	s1440p			No viewfinder still image capture
------	--------	--	--	-----------------------------------

YUV / PRIV	s720p	JPEG	s1440p	Standard still imaging
------------	-------	------	--------	------------------------

YUV / PRIV	s720p	YUV / PRIV	s1440p	In-app video or processing with preview
------------	-------	------------	--------	---

Devices with the MONOCHROME capability

(`CameraCharacteristics#REQUEST_AVAILABLE_CAPABILITIES` includes

`CameraMetadata#REQUEST_AVAILABLE_CAPABILITIES_MONOCHROME`) supporting Y8 must support substituting YUV streams with Y8 in all guaranteed stream combinations.

s720p refers to 720p (1280 x 720) or the maximum supported resolution for the particular format returned by `StreamConfigurationMap.getOutputSizes()`. s1440p refers to 1440p (1920 x 1440) or the maximum supported resolution for the particular format returned by `StreamConfigurationMap.getOutputSizes()`. Devices whose capabilities don't include `ANDROID_REQUEST_AVAILABLE_CAPABILITIES_BACKWARD_COMPATIBLE` must support at least a single Y16 stream, Dataspace::DEPTH with sVGA resolution, during concurrent operation, where sVGA is the smaller of the two following resolutions:

maximum output resolution for the given format

640 x 480

Implementation

To allow apps to query a device to determine if its cameras support concurrent streaming,

implement the ICameraProvider@2.6 HAL interface, which includes the following methods:

`getConcurrentStreamingCameraIds`
`isConcurrentStreamCombinationSupported`

For a reference implementation of the ICameraProvider@2.6HAL interface, see the emulated camera HAL library at `EmulatedCameraProviderHWLImpl.cpp`.

Validation

To test that your implementation of this feature works as intended, use the `ConcurrentCameraTest.java` CTS test. Also, test using an app that opens up multiple cameras and operates them concurrently.

Resource allocations problems

If camera HALs advertise support for concurrent operation of camera devices, they might run into resource allocation problems, especially in the case where there are enough image signal processor (ISP) resources on the phone to stream both front and back (or other) cameras concurrently, but not to their full capacity. In this case, the camera HAL must allocate limited hardware resources to each camera device.

Example scenario

The following scenario demonstrates this problem.

Problem

The device has the following configuration:

Camera ID 0 is a logical camera backed by a wide and ultra-wide camera, which each take one ISP resource.

Camera ID 1 is a camera which takes one ISP resource.

The device (phone) has two ISPs. If camera ID 0 is opened and a session is configured, it's possible that the camera HAL reserves two ISPs anticipating both ultrawide and wide camera use.

If that's the case, the front camera (ID 1) can't configure any streams because both ISPs are in use.

Solution

To address this problem, the framework can open both camera IDs 0 and 1 before configuring sessions to provide a hint to the camera HAL about how to allocate resources (because it now expects concurrent operation of cameras). However, this can lead to limited capabilities, for example, zoom might not be able to handle the full zoom range ratio (because switching physical camera IDs might be problematic).

To implement this solution, make the following updates to `provider@2.6::ICameraProvider::getConcurrentCameraStreamingCameraIds`.

Mandate that for concurrent operation of cameras, the camera framework must open camera

devices (@3.2::ICameraDevice::open) before configuring any sessions on the camera devices. This allows camera providers to allocate resources accordingly.

To address the issue of not being able to handle the full zoom range ratio, ensure that camera apps, when using cameras concurrently, are guaranteed to use the ZOOM_RATIO control setting between only 1x and MAX_DIGITAL_ZOOM instead of the complete ZOOM_RATIO_RANGE (this prevents the switching of physical cameras internally, which potentially requires more ISPs).

Note: Camera IDs advertised in a combination through getConcurrentStreamingCameraIds must not conflict. For example if getConcurrentStreamingCameraIds advertises {0,1}, {2,1}, camera IDs 0 and 2 are allowed to conflict, however camera IDs 0 and 1, and 1 and 2 must not conflict.

Problem with testDualCameraPreview

When you make the updates above, it can create a problem with a behavior allowed by the MultiViewTest.java#testDualCameraPreview test.

The test testDualCameraPreview doesn't configure sessions only after opening all cameras. It follows this sequence:

for each camera in cameraDevices :

```
device = openCamera(camera)
createCaptureSession(device);
```

It does, however, tolerate camera open failures with ERROR_MAX_CAMERAS_IN_USE [1]. Third-party apps might depend on this behavior.

Because the camera HAL won't know the complete set of camera IDs being opened for concurrent operation before configuring sessions, it could be hard for it to allocate hardware resources (assuming there is some competition for them).

To address this problem, maintaining backward compatibility in addition to supporting concurrent streaming, camera HALs should fail openCamera calls with ERROR_MAX_CAMERAS_IN_USE if they can't support full stream configuration for all cameras running concurrently.

Here's a **simple explanation of Concurrent Camera Streaming** in Android:

What is Concurrent Camera Streaming?

It means **using two or more cameras at the same time** — for example, front and back cameras capturing simultaneously in an app.

Think of:

- Picture-in-picture style selfies + rear view.
- AR, fitness, or video call apps showing multiple views.

Support in Android

Since **Android 11**, the Camera2 API allows apps to:

1. **Check which cameras can be used together (`getConcurrentCameraIds`)**
2. **Check if a specific session config is allowed (`isConcurrentSessionConfigurationSupported`)**

The device must tell Android what camera combinations and stream types it supports.

Required Stream Support

For each pair of concurrent cameras, the device **must support these stream types**:

Stream 1	Max size	Stream 2	Max size	Use Case
YUV	1440p	—	—	Processing or filters
PRIVATE	1440p	—	—	Viewfinder
JPEG	1440p	—	—	Capture without preview
YUV/PRIVATE	720p	JPEG	1440p	Preview + still capture
YUV/PRIVATE	720p	YUV/PRIVATE	1440p	Dual preview or processing

 *Y8 (monochrome) or Y16 (depth) must also be supported in some special cases.*

HAL Implementation

The device maker must implement HAL version **ICameraProvider@2.6** and provide:

- **`getConcurrentStreamingCameraIds()`**
→ tells the framework which camera pairs can be run together
- **`isConcurrentStreamCombinationSupported()`**
→ checks if a stream setup will work before trying

Also, update HAL logic to **allocate ISP resources smartly** when concurrent use is expected.

Resource Conflict Example

Problem:

- Camera 0 (ultrawide + wide) uses 2 ISPs

- Camera 1 (front) needs 1 ISP
- But only 2 ISPs are available → can't open both

Solution:

- Framework should open **all needed cameras before setting up sessions**
- HAL can then share or reserve ISP resources properly

Validation & Compatibility

- Run CTS: `ConcurrentCameraTest.java`
- Test with apps that use multiple cameras (e.g., dual-view)
- Maintain backward compatibility: If resource is short, **fail openCamera() with ERROR_MAX_CAMERAS_IN_USE**, not silently degrade.

Summary

Concept	Explanation
Purpose	Use multiple cameras at the same time
APIs	<code>getConcurrentCameraIds, isConcurrentSessionConfigurationSupported</code>
Device Requirement	Must support defined stream resolutions & formats
HAL Version Needed	<code>ICameraProvider@2.6</code>
Testing	CTS + real app testing
ISP Resource Conflict	Handled by opening cameras <i>before</i> sessions are created
Backward Compatibility	Fail gracefully if limits are hit

Would you like help with HAL code structure or framework-side logic for this?



⚠ Resource Conflict Example **Problem:** * Camera 0 (ultrawide + wide) uses 2 ISPs * Camera 1 (front) needs 1 ISP * But only 2 ISPs are available → can't open both **Solution:** * Framework should open **all needed cameras before setting up sessions** * HAL can then share or reserve ISP resources properly
explain

Let's break down the **resource conflict example** in **simple terms**:

Goal

You want to **open two cameras at the same time** — for example:

- Camera 0: a rear module with **two lenses** (ultrawide and wide)
- Camera 1: a **front-facing camera**

Hardware Limitation: ISP

ISP = Image Signal Processor

Each camera lens needs an ISP to process raw image data (e.g., denoising, color correction).

Imagine your device has only **2 ISPs**.

The Problem

Camera	Lenses	ISP Needed
Camera 0	Ultrawide + Wide	2 ISPs
Camera 1	Front	1 ISP

You want to open both cameras **at the same time**, but you need **3 ISPs**, and the device only has **2**.

→ **Not enough hardware resources.**

🔧 Naive Attempt (What not to do)

If the app does:

```
java
cameraManager.openCamera("0")
cameraManager.openCamera("1")
```

And **each call tries to grab resources immediately**, you'll hit a problem:

- Camera 0 grabs **both ISPs**.
- Camera 1 tries to open but **fails** because no ISP is left.

✓ The Right Solution: Deferred Configuration

Let's say Android framework and HAL **wait** before locking ISPs.

Steps:

1. App **opens both cameras** first (`openCamera()`).
 - Framework **delays resource allocation**.
2. App **creates sessions** (`createCaptureSession()`).
3. HAL now sees both camera requests **together**, and can **balance or share** ISPs.

HAL can now:

- Allocate only **1 ISP** to Camera 0 (disable one lens).
- Allocate 1 ISP to Camera 1.
- Inform the framework of a working configuration.

→ Result: **Both cameras work**, maybe with some limitation (like 1 lens active in Camera 0).

Analogy: Restaurant Table Booking

Imagine:

- Restaurant has 2 tables (ISPs)
- You have 2 friend groups (cameras) needing tables
- One group alone books both tables — the second can't sit

Better way: Wait for **both groups to arrive**, then assign tables efficiently — maybe one group shares a table

Key Point

The **Android framework must open all cameras before starting sessions**, so that the **HAL can make smarter resource decisions** — not just give everything to the first camera.

Would you like a visual of this flow or code references from the framework side?



Multi-camera support

`bookmark_border`

Android 9 introduced API support for multi-camera devices through a new logical camera device composed of two or more physical camera devices pointing in the same direction. The logical camera device is exposed as a single `CameraDevice/CaptureSession` to an app allowing

for interaction with HAL-integrated multi-camera features. Apps can optionally access and control underlying physical camera streams, metadata, and controls.

Multi-camera support

Figure 1. Multi-camera support

In this diagram, different camera IDs are color coded. The app can stream raw buffers from each physical camera at the same time. It is also possible to set separate controls and receive separate metadata from different physical cameras.

Examples and sources

Multi-camera devices must be advertised with the logical multi-camera capability.

Camera clients can query the camera ID of the physical devices a particular logical camera is made of by calling `getPhysicalCameraIds()`. The IDs returned as part of the result are then used to control physical devices individually through `setPhysicalCameraId()`. The results from such individual requests can be queried from the complete result by invoking `getPhysicalCameraResults()`.

Individual physical camera requests may support only a limited subset of parameters. To receive a list of the supported parameters, developers can call `getAvailablePhysicalCameraRequestKeys()`.

Physical camera streams are supported only for non-reprocessing requests and only for monochrome and bayer sensors.

Implementation

Support checklist

To add logical multi-camera devices on the HAL side:

Add a `ANDROID_REQUEST_AVAILABLE_CAPABILITIES_LOGICAL_MULTI_CAMERA` capability for any logical camera device backed by two or more physical cameras that are also exposed to an app.

Populate the static `ANDROID_LOGICAL_MULTI_CAMERA_PHYSICAL_IDS` metadata field with a list of physical camera IDs.

Populate the depth-related static metadata required to correlate between physical camera streams' pixels: `ANDROID_LENS_POSE_ROTATION`, `ANDROID_LENS_POSE_TRANSLATION`, `ANDROID_LENS_INTRINSIC_CALIBRATION`, `ANDROID_LENS_DISTORTION`, `ANDROID_LENS_POSE_REFERENCE`.

Set the static `ANDROID_LOGICAL_MULTI_CAMERA_SENSOR_SYNC_TYPE` metadata field to:

`ANDROID_LOGICAL_MULTI_CAMERA_SENSOR_SYNC_TYPE_APPROXIMATE`: For sensors in main-main mode, no hardware shutter/exposure sync.

`ANDROID_LOGICAL_MULTI_CAMERA_SENSOR_SYNC_TYPE_CALIBRATED`: For sensors in main-secondary mode, hardware shutter/exposure sync.

Populate ANDROID_REQUEST_AVAILABLE_PHYSICAL_CAMERA_REQUEST_KEYS with a list of supported parameters for individual physical cameras. The list can be empty if the logical device doesn't support individual requests.

If individual requests are supported, process and apply the individual physicalCameraSettings that can arrive as part of capture requests and append the individual physicalCameraMetadata accordingly.

For Camera HAL device versions 3.5 (introduced in Android 10) or higher, populate the ANDROID_LOGICAL_MULTI_CAMERA_ACTIVE_PHYSICAL_ID result key using the ID of the current active physical camera backing the logical camera.

For devices running Android 9, camera devices must support replacing one logical YUV/Raw stream with physical streams of the same size (doesn't apply to Raw streams) and the same format from two physical cameras. This doesn't apply to devices running Android 10.

For devices running Android 10 where the camera HAL device version is 3.5 or higher, the camera device must support isStreamCombinationSupported for apps to query whether a particular stream combination containing physical streams is supported.

Stream configuration map

For a logical camera, the mandatory stream combinations for the camera device of a certain hardware level is the same as what's required in CameraDevice.createCaptureSession. All of the streams in the stream configuration map must be logical streams.

For a logical camera device supporting Raw capability with physical sub-cameras of different sizes, if an app configures a logical Raw stream, the logical camera device must not switch to physical sub-cameras with different sensor sizes. This ensures that existing Raw capture apps don't break.

To take advantage of HAL-implemented optical zoom by switching between physical sub-cameras during Raw capture, apps must configure physical sub-camera streams instead of a logical Raw stream.

Guaranteed stream combination

Both the logical camera and its underlying physical cameras must guarantee the mandatory stream combinations required for their device levels.

A logical camera device should operate in the same way as a physical camera device based on its hardware level and capabilities. It's recommended that its feature set is a superset of that of individual physical cameras.

On devices running Android 9, for each guaranteed stream combination, the logical camera must support:

Replacing one logical YUV_420_888 or raw stream with two physical streams of the same size

and format, each from a separate physical camera, given that the size and format are supported by the physical cameras.

Adding two raw streams, one from each physical camera, if the logical camera doesn't advertise RAW capability, but the underlying physical cameras do. This usually occurs when the physical cameras have different sensor sizes.

Using physical streams in place of a logical stream of the same size and format. This must not slow down the frame rate of the capture when the minimum frame duration of the physical and logical streams are the same.

Performance and power considerations

Performance:

Configuring and streaming physical streams may slow down the logical camera's capture rate due to resource constraints.

Applying physical camera settings may slow down the capture rate if the underlying cameras are put into different frame rates.

Power:

HAL's power optimization continues to work in the default case.

Configuring or requesting physical streams may override HAL's internal power optimization and incur more power use.

Customization

You can customize your device implementation in the following ways.

The fused output of the logical camera device depends entirely on the HAL implementation. The decision on how fused logical streams are derived from the physical cameras is transparent to the app and Android camera framework.

Individual physical requests and results can be optionally supported. The set of available parameters in such requests is also entirely dependent on the specific HAL implementation.

From Android 10, the HAL can reduce the number of cameras that can be directly opened by an app by electing not to advertise some or all PHYSICAL_IDs in getCameraIdList. Calling getPhysicalCameraCharacteristics must then return the characteristics of the physical camera.

Validation

Logical multi-camera devices must pass camera CTS like any other regular camera. The test cases that target this type of device can be found in the LogicalCameraDeviceTest module.

These three ITS tests target multi-camera systems to facilitate the proper fusing of images:

`scene1/test_multi_camera_match.py`

`scene4/test_multi_camera_alignment.py`

`sensor_fusion/test_multi_camera_frame_sync.py`

The scene 1 and scene 4 tests run with the ITS-in-a-box test rig. The `test_multi_camera_match` test asserts that the brightness of the center of the images match when the two cameras are both enabled. The `test_multi_camera_alignment` test asserts that camera spacings, orientations,

and distortion parameters are properly loaded. If the multi-camera system includes a Wide FoV camera (>90°), the rev2 version of the ITS box is required.

Sensor_fusion is a second test rig that enables repeated, prescribed phone motion and asserts that the gyroscope and image sensor timestamps match and that the multi-camera frames are in sync.

All boxes are available through AcuSpec, Inc. (www.acuspecinc.com, fred@acuspecinc.com) and MYWAY Manufacturing (www.myway.tw, sales@myway.tw). Additionally, the rev1 ITS box can be purchased through West-Mark (www.west-mark.com, dgoodman@west-mark.com).

Best practices

To fully take advantage of features enabled by multi-camera while maintaining app compatibility, follow these best practices when implementing a logical multi-camera device:

(Android 10 or higher) Hide physical sub-cameras from `getCameraIdList`. This reduces the number of cameras that can be directly opened by apps, eliminating the need for apps to have complex camera selection logic.

(Android 11 or higher) For a logical multi-camera device supporting optical zoom, implement the `ANDROID_CONTROL_ZOOM_RATIO` API, and use `ANDROID_SCALER_CROP_REGION` for aspect ratio cropping only. `ANDROID_CONTROL_ZOOM_RATIO` enables the device to zoom out and maintain better precision. In this case, the HAL must adjust the coordinate system of `ANDROID_SCALER_CROP_REGION`, `ANDROID_CONTROL_AE_REGIONS`, `ANDROID_CONTROL_AWB_REGIONS`, `ANDROID_CONTROL_AF_REGIONS`, `ANDROID_STATISTICS_FACE_RECTANGLES`, and `ANDROID_STATISTICS_FACE_LANDMARKS` to treat the post-zoom field of view as the sensor active array. For more information on how `ANDROID_SCALER_CROP_REGION` works together with `ANDROID_CONTROL_ZOOM_RATIO`, see [camera3_crop_reprocess#cropping](#).

For multi-camera devices with physical cameras that have different capabilities, make sure the device advertises support for a certain value or range for a control only if the whole zoom range supports the value or range. For example, if the logical camera is composed of an ultrawide, a wide, and a telephoto camera, do the following:

If the active array sizes of the physical cameras are different, the camera HAL must do the mapping from the physical cameras' active arrays to the logical camera active array for `ANDROID_SCALER_CROP_REGION`, `ANDROID_CONTROL_AE_REGIONS`, `ANDROID_CONTROL_AWB_REGIONS`, `ANDROID_CONTROL_AF_REGIONS`, `ANDROID_STATISTICS_FACE_RECTANGLES`, and `ANDROID_STATISTICS_FACE_LANDMARKS` so that from the app's perspective, the coordinate system is the logical camera's active array size.

If the wide and telephoto cameras support autofocus, but the ultrawide camera is fixed focus, make sure the logical camera advertises autofocus support. The HAL must simulate an autofocus state machine for the ultrawide camera so that when the app zooms out to the ultrawide lens, the fact that the underlying physical camera is fixed focus is transparent to the app, and the autofocus state machines for the supported AF modes work as expected.

If the wide and telephoto cameras support 4K @ 60 fps, and the ultrawide camera only supports 4K @ 30 fps, or 1080p @ 60 fps, but not 4K @ 60 fps, make sure the logical camera doesn't advertise 4k @ 60 fps in its supported stream configurations. This guarantees the integrity of

the logical camera capabilities, ensuring that the app won't run into the issue of not achieving 4k @ 60 fps at a ANDROID_CONTROL_ZOOM_RATIO value of less than 1.

Starting from Android 10, a logical multi-camera isn't required to support stream combinations that include physical streams. If the HAL supports a combination with physical streams: (Android 11 or higher) To better handle use cases such as depth from stereo and motion tracking, make the field of view of the physical stream outputs as large as can be achieved by the hardware. However, if a physical stream and a logical stream originate from the same physical camera, hardware limitations might force the field of view of the physical stream to be the same as the logical stream.

To address the memory pressure caused by multiple physical streams, make sure apps use discardFreeBuffers to deallocate the free buffers (buffers that are released by the consumer, but not yet dequeued by the producer) if a physical stream is expected to be idle for a period of time.

If physical streams from different physical cameras aren't typically attached to the same request, make sure apps use surface group so that one buffer queue is used to back two app-facing surfaces, reducing memory consumption.

Here's a **clear explanation of multi-camera support** in Android, especially for logical multi-camera devices (introduced in Android 9+):

What Is a Logical Multi-Camera?

A **logical camera** is:

- A **single camera ID** (like "0") exposed to apps,
- But **internally backed by multiple physical cameras** (like wide + telephoto),
- Apps see just one camera, but the HAL can **switch between or fuse** physical cameras.

 Purpose:

- Enable features like **optical zoom, depth sensing, low-light fusion, super-resolution**, etc.,
- **Simplify app logic:** Apps don't need to know which lens is in use.

What Apps Can Do

Apps use the **Camera2 API** and can:

- Stream from the **logical camera** as usual.
- Optionally access **individual physical camera IDs** (via `getPhysicalCameraIds()`).
- Set separate controls per physical camera with `setPhysicalCameraId()`.
- Get individual results from `getPhysicalCameraResults()`.

What HAL (OEMs) Must Do

To support logical multi-camera:

1. Advertise capabilities

- Set `LOGICAL_MULTI_CAMERA` in `ANDROID_REQUEST_AVAILABLE_CAPABILITIES`
- Fill in `ANDROID_LOGICAL_MULTI_CAMERA_PHYSICAL_IDS` with actual camera IDs
- Provide calibration data like:
 - `LENS_POSE_ROTATION`
 - `LENS_POSE_TRANSLATION`
 - `LENS_INTRINSIC_CALIBRATION`
 - `LENS_DISTORTION`
- Set `SENSOR_SYNC_TYPE` to either:
 - `APPROXIMATE` (no shutter sync)
 - `CALIBRATED` (with shutter sync)

2. Control physical streams (optional)

- If supported, provide:
 - `AVAILABLE_PHYSICAL_CAMERA_REQUEST_KEYS`
 - Handle `physicalCameraSettings` in capture requests
 - Append `physicalCameraMetadata` in results

3. Expose only logical camera (best practice)

- From Android 10+, **don't expose physical camera IDs** in `getCameraIdList()` to simplify app logic.

Validation (Testing)

To ensure correct behavior:

- Run CTS tests: `LogicalCameraDeviceTest`
- Run ITS (Image Test Suite) tests:
 - `test_multi_camera_match.py`: brightness match
 - `test_multi_camera_alignment.py`: lens alignment and calibration

- `test_multi_camera_frame_sync.py`: shutter sync with motion
-

Best Practices for Implementation

Zoom:

- Use `ZOOM_RATIO` (Android 11+) for zoom control.
- Let HAL handle physical camera switching internally.
- Apps just zoom smoothly — don't know or care which lens is active.

Exposure & Regions:

If the app sets:

- AE / AF / AWB regions
- Face detection regions

Then HAL must:

- Map these to the **logical camera coordinate space**, even if they apply to a physical sub-camera.

Autofocus:

- If only some lenses support AF, HAL must **simulate AF state machine** when using fixed-focus lenses (e.g., ultrawide).

Stream Config:

If:

- One camera supports 4K @ 60 fps, but another doesn't,
- Don't advertise 4K @ 60 unless it's **supported throughout the full zoom range**.

Memory Optimization:

- Use `discardFreeBuffers()` if a physical stream goes idle.
 - Use `surfaceGroup` to share buffers across app-facing surfaces.
-

Physical Streams (Advanced)

Apps **can request** physical streams from logical camera if HAL supports it. Use cases:

- Stereo depth sensing
- Multi-view recording

If supported:

- Use `isStreamCombinationSupported()` (Android 10+)
- Make sure frame rate doesn't degrade when using physical streams instead of logical ones

🎯 Summary

Feature	Logical Camera	Physical Cameras
Camera ID	Single (e.g., "0")	Multiple ("0_wide", "0_tele")
App access	Simple by default	Optional via advanced API
HAL behavior	Fuses input, switches lenses	Exposes control if supported
Benefit	Optical zoom, better low light, seamless UX	Depth, stereo, advanced imaging

Would you like a **diagram** or a **code walkthrough** to show how to work with logical + physical cameras in Camera2 API?



explain in details in simple words

Let's break **Android Camera Bokeh feature** down step by step in **simple terms**.

🔍 What is Bokeh?

Bokeh is a photography effect where:

- The subject is in focus (sharp).
- The **background is blurred** to give a "depth" feel.

Think of portrait photos where the person is clear, but the background looks soft and dreamy.

💡 How Do Phones Create Bokeh?

Mobile phones use **depth information** to decide what to blur:

1. **Dual Cameras:** Like human eyes, two cameras can "see" depth.
2. **Dual Photodiodes (PD):** A single sensor with two light sensors per pixel can also estimate depth.

This depth info is used to **blur the background** while keeping the subject sharp.

Android Support

Starting with **Android 11**, Android provides **standard APIs** so all apps can **use bokeh** if the hardware supports it.

Implementation Steps

If you're a **device manufacturer** or **HAL implementer**, you need to:

1. Declare Supported Bokeh Modes

Declare in the `CameraCharacteristics` metadata:

```
cpp  
  
ANDROID_CONTROL_AVAILABLE_EXTENDED_SCENE_MODE_MAX_SIZES
```

This is a list of tuples like:

```
{mode, maxWidth, maxHeight}
```

Example:

```
yaml  
  
{BOKEH_STILL_CAPTURE, 1920, 1080}  
{BOKEH_CONTINUOUS, 1280, 720}
```

These tell the system what bokeh modes are available and what maximum resolution each can support.

2. Declare Zoom Support

In:

```
cpp
```



How logical camera works

Great question!

What is a "Logical Camera" in Android?

In **simple words**, a **logical camera** is a **virtual camera device** that Android presents to apps, which **combines multiple physical cameras** under the hood.

Why Use Logical Cameras?

Modern phones often have:

-  A main (wide) camera
-  A telephoto camera
-  An ultra-wide camera
-  A night/monochrome sensor
-  A depth sensor

Instead of showing each of these separately to apps, Android can combine them into one "logical" camera.

This gives the app a **unified experience**, while the system chooses the best physical camera for the job.

How Does It Work?

Under the hood:

Component	Role
Logical camera	The camera ID apps use (e.g. "0"), exposed by the Camera HAL .
Physical cameras	The real sensors that the logical camera wraps together.
Camera HAL	Decides how to route requests to physical cameras based on zoom, light, etc.

Example:

App opens camera "0" (logical), but based on zoom level:

- Zoom 1x → main sensor
- Zoom 2x → telephoto sensor
- Zoom 0.5x → ultra-wide sensor

Android switches **automatically**, without app needing to know.

Key Metadata Fields

- **ANDROID_REQUEST_AVAILABLE_CAPABILITIES_LOGICAL_MULTI_CAMERA**
→ Means this is a logical camera.
- **ANDROID_LOGICAL_MULTI_CAMERA_PHYSICAL_IDS**
→ Lists the physical camera IDs it wraps (e.g., "0", "2").
- **CameraCharacteristics.getPhysicalCameraIds()**
→ Apps can get this info (starting Android 10).

Can Apps Access Physical Cameras?

Yes! Since Android 10 (API 29), apps can:

- Use **logical camera** for simplicity.
- Or directly request capture from **specific physical camera** for advanced use.

Using:

```
java
CameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_PREVIEW)
    .addTarget(surface)
    .setPhysicalCameraId("2"); // Access telephoto directly
```

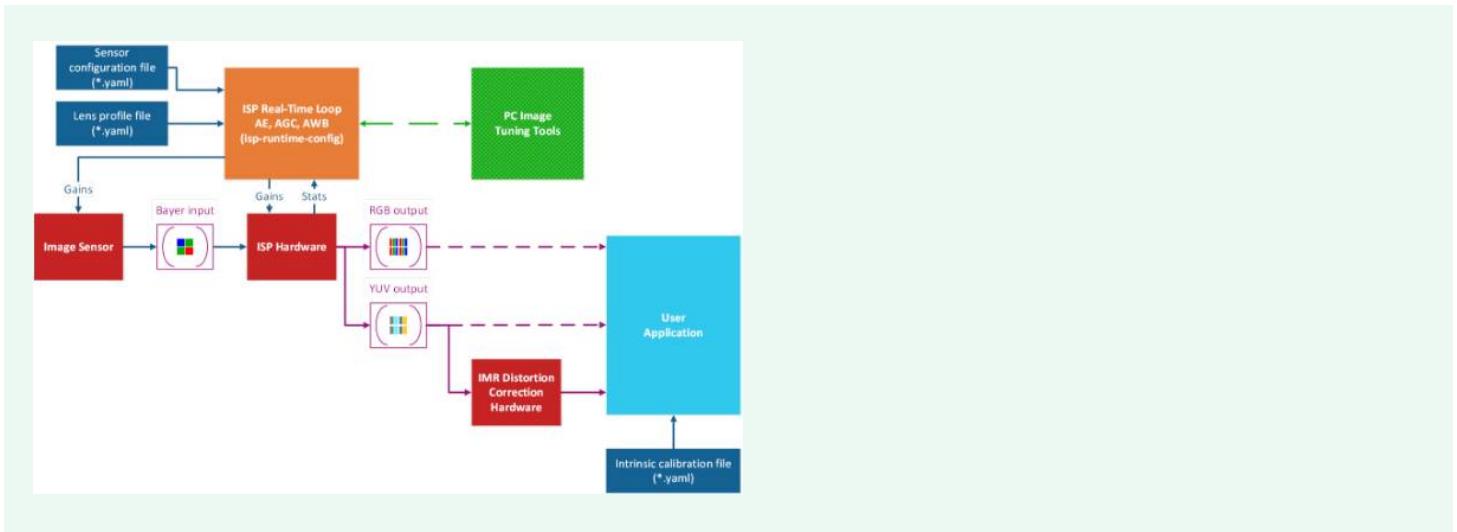
Use Case Examples

Scenario	Logical Camera Use
Zooming during video	Switches physical cameras behind the scenes
Low light capture	Chooses best physical sensor for light
Bokeh effect	Combines RGB + depth sensor
Single app camera interface	App sees one camera, not 3 or 4

Summary

Term	Meaning
Logical Camera	Virtual camera that wraps multiple real sensors
Physical Camera	Actual hardware sensor on the phone
Benefit	Simpler for apps, more flexibility for manufacturers

Let me know if you want a **diagram** to visualize this or want to see **real examples from AOSP**.



This diagram shows a **typical ISP (Image Signal Processor) pipeline**, which takes raw image data from a sensor and processes it into usable image formats like YUV or RGB for apps to use.

Step-by-Step Explanation in Simple Words

1. Image Sensor (Red Box)

- This is the **physical camera sensor** that captures raw light information.
- Output is usually in **Bayer format** (a grid of red, green, and blue pixels).

2. Gains from Configuration

- Sensor config and lens profile files (`*.yaml`) define **initial tuning**, like exposure, lens corrections, etc.
- These gains (brightness adjustments) go to both:
 - The **sensor**
 - The **ISP Hardware**

3. ISP Hardware (Image Signal Processor)

- This is the **core processor** that turns raw Bayer data into usable images.
- Tasks include:
 - **Demosaicing** (converting Bayer to RGB/YUV)
 - **Noise reduction**

- **Sharpening**
 - **Color correction**
 - Outputs:
 - **RGB output**
 - **YUV output**
 - Sends **stats** back to ISP Real-Time Loop (for tuning)
-

4. ISP Real-Time Loop (Orange Box)

- Performs **auto exposure (AE)**, **auto gain (AGC)**, and **auto white balance (AWB)** dynamically.
 - Uses the **isp-runtime-config** and runtime **stats** to adjust gains on the fly.
 - Also works with **PC tuning tools** (green box) during development.
-

5. PC Image Tuning Tools (Green Box)

- Used during development to adjust camera tuning profiles.
 - Helps optimize AE, AWB, and image quality before deploying to real devices.
-

6. YUV → IMR Distortion Correction (Optional)

- If needed, YUV output can be passed through **IMR hardware** to correct lens distortion (e.g., wide-angle bending).
 - Uses an **intrinsic calibration file** to apply correct math.
-

7. User Application (Blue Box)

- The final RGB or YUV data (corrected or raw) is passed to your app (like camera, AR app, etc.).
 - App doesn't need to know the whole pipeline — just gets a clean image frame.
-

Summary Table

Stage	Purpose
Sensor	Captures raw image (Bayer format)
ISP	Converts Bayer to RGB/YUV, applies corrections
Real-Time Loop	Auto-exposure, white balance, dynamic tuning
IMR	Optional lens distortion correction

Stage	Purpose
App	Receives clean image frame

Let me know if you want a version of this with **labels overlaid** or a **flowchart-style simplification!**



When it comes to automotive cameras, customer expectations are high. Familiar with the performance of relatively inexpensive consumer products, customers expect an equally impressive HD view from their vehicle's backup camera.

These high customer expectations can only be met with Megapixel cameras. Previously, those contained a CMOS imager chip and a separate image signal processing (ISP) chip.

The cost of a separate ISP chip can be avoided by moving the image signal processing instead into the infotainment headunit or the surround view module. There, an existing processor can perform the ISP function without incremental hardware cost.

You're absolutely right — let's break this down in **simple terms** so it's clear:



Automotive Cameras and High Expectations

- Today's **car buyers expect clear, high-definition (HD) images** from cameras like:
 - Rear-view (backup) cameras
 - Surround-view (bird's-eye) systems
- This expectation is shaped by **smartphone and consumer electronics**, which offer great image quality even at low cost.



Why Megapixel Cameras Are Needed

- To meet HD expectations, automotive systems need **Megapixel image sensors** (not old VGA-quality ones).
- A **Megapixel camera** captures **more detail**, which is critical for:
 - Seeing objects clearly while reversing
 - Parking assistance
 - Detecting pedestrians or lanes in ADAS (Advanced Driver Assistance Systems)



The Old Way: Separate ISP Chip

- Traditionally, each camera module had:
 - A **CMOS sensor** (captures the image)
 - A **separate ISP chip** (cleans up and processes the raw image)
- But... this adds **extra cost**, size, and heat.

The New Way: Centralized ISP in the Headunit

- Instead of putting an ISP chip in **every camera**, we move image processing to a **central computer** in the car:
 - The **infotainment head unit**
 - Or the **surround view module**
- These already have powerful CPUs/GPUs, so:
 - They can run ISP software
 - No need for extra hardware
 - **Lower total system cost**

Benefits

Benefit	Explanation
 Cheaper	No separate ISP chip in each camera module
 More flexible	Central software can be updated or tuned easily
 Easier integration	Fewer components per camera, simpler hardware
 Scalable	Add more cameras without multiplying chip cost

Summary

Old way = Sensor + ISP chip in each camera (expensive)

New way = Sensor only in camera, processing done centrally (cost-efficient)

So, centralizing ISP in the headunit is like moving from lots of tiny brains to one big shared brain — smart, efficient, and cheaper.

Let me know if you want diagrams or automotive-specific examples (e.g., backup camera vs surround view vs driver monitoring).



IMR

explain

Certainly! Let's break down **IMR (Image Memory Remapper)** in **simple words**, especially in the context of automotive image pipelines and ISPs.

What is IMR (Image Memory Remapper)?

IMR = Image Memory Remapper

It is a **hardware block** (or sometimes a software function) that is used inside an **Image Signal Processor (ISP)** or a **SoC** (System on Chip).

Why do we need IMR?

Raw image data from a camera sensor doesn't always come out in the format or shape we want.

- The image may be:
 - Cropped incorrectly
 - Rotated (e.g., 90°, 180°)
 - Mirrored
 - Stored in **non-continuous memory blocks** (tiled or planar layout)
 - In a **distorted shape** due to wide-angle lens or sensor layout

 **IMR helps reformat, reposition, or remap this image data** into a usable form for display or processing.

What does IMR do exactly?

Function	Description
Remapping	Reorders or rearranges pixels in memory
Cropping / Resizing	Changes image size or trims unnecessary areas
Rotation / Flip	Rotates or mirrors image (useful for rear cameras)
Address Translation	Deals with complex memory layouts or buffer shapes
Format conversion	May convert from planar to interleaved formats (e.g., YUV420 to RGB)

Example: Automotive Use Case

Let's say you have a **rear-view camera**:

- Mounted upside down
- Outputs a raw YUV image
- Image is mirrored due to lens setup
- Image is stored in separate memory tiles (not linear)

 Before the display or ADAS algorithm can use it, we must:

1. Rotate the image
2. Mirror it
3. Reassemble tiled data into a flat buffer

 The **IMR handles all these operations efficiently** — often using DMA or hardware accelerators — without burdening the CPU/GPU.

Think of it like this:

The **IMR is a smart librarian** for image memory:

- Takes the messy shelves (raw data)
- Reorders them
- Rotates the pages
- Stacks them into a neat book
- Hands it off to the next stage (display or ML model)

Summary

Feature	IMR Purpose
 Format conversion	Reshape image buffers
 Rotation/Flip	Correct orientation
 Memory remap	Convert between tiled and linear formats
 Crop/resize	Fit display or processing needs

So, **IMR ensures your camera data is correctly shaped and placed in memory**, making it ready for display, processing, or machine learning inference — crucial in automotive safety systems.

Let me know if you want a visual example (before vs after memory remap) or how it's used in Linux/V4L2 or embedded systems!



ISP Pipeline

Architecturally, moving ISP from the camera to the headunit or surround view module simply makes sense. Execution, however, can be tricky and less straight forward than one might expect. This is where Cogent's experience and existing code may help: We have been working with automotive cameras for many years and provide an end-to-end solution that addresses the pitfalls of implementing ISP on existing hardware.

Our embedded code is the real-time loop that controls the camera to provide a perfect view around the vehicle, no matter how bright or dark the environment. It performs the following functions:

Raw data correction (remove defective pixels)

Lens correction (correct geometric and luminance/color distortions)

Noise reduction (Apply temporal and/or spatial averaging to increase SNR (signal to noise ratio).

Demosaic (reconstruct colors per pixel)

Exposure control

White balance

Color correction (adjust color in different lighting conditions, including challenging low light conditions)

Noise reduction

Color space conversion: RGB or YUV output

Great! Let's walk through this **ISP pipeline** step-by-step in **very simple words**, with a **focus on automotive camera systems**, especially when the **ISP is moved from inside the camera to a centralized module** like a **head unit** or **surround view ECU**.



What is an ISP (Image Signal Processor)?

An **ISP** is like a **mini factory** that takes the **raw image** from a camera sensor and **cleans it up, colors it, and gets it ready to be shown or processed** (e.g., for object detection).

Instead of doing this in the camera hardware, automakers often move the ISP into a **central ECU (Electronic Control Unit)** — for example, the **head unit** or the **surround view module**. This allows better control and coordination of multiple cameras.



Why move ISP from the camera to the ECU?

Advantages:

- You have full control of the image processing pipeline

- You can **standardize** and **update** image processing logic across different cameras
- It's easier to **fuse** images from multiple cameras (surround view, ADAS)
- Centralized processing may be more **power-efficient** or **cost-effective**

⚠ Challenges:

- Raw image data is large (more bandwidth needed)
- Real-time constraints are tighter (must process fast!)
- Tuning ISP software on generic SoCs is **non-trivial**

🧠 What does the ISP pipeline do? (Step-by-step)

This is what Cogent's pipeline handles:

1. 🔧 Raw Data Correction

- **Problem:** Sensors can have **dead** or **stuck pixels**.
- **Fix:** ISP identifies and replaces defective pixels using neighbors.
- **Think:** Like patching a bad pixel on your TV.

2. 🎯 Lens Correction

- **Problem:** Lenses bend light in weird ways — especially wide-angle ones.
- **Fix:** ISP straightens bent edges (geometry) and corrects uneven brightness/color across the image.
- **Example:** Removes “fisheye” look and fixes corner darkness (vignetting).

3. 📷 Noise Reduction (First pass)

- **Problem:** In low light, image looks grainy.
- **Fix:** ISP applies **spatial (per frame)** and/or **temporal (across frames)** filtering to clean it.
- **Goal:** Make the image smooth without losing detail.

4. 🎨 Demosaic

- **Problem:** Camera sensor only captures **one color (R, G, or B)** per pixel using a **Bayer pattern**.
- **Fix:** ISP fills in missing colors using smart algorithms.
- **Result:** Full RGB image from raw Bayer.

5. ☼ Exposure Control

- **Problem:** Image too dark or too bright depending on lighting.
- **Fix:** ISP adjusts gain and exposure time automatically.

6. ⚪ White Balance

- **Problem:** White objects may look yellowish (sunlight) or blueish (LED lights).
- **Fix:** ISP adjusts color gain so that white looks white.
- **Adapts** to different lighting environments.

7. 🎨 Color Correction

- **Problem:** Even with good lighting, colors may still be inaccurate.
- **Fix:** ISP uses a **Color Correction Matrix (CCM)** and possibly LUTs (Look-Up Tables) to fine-tune color appearance.
- **Bonus:** Helps in difficult scenes like tunnels or headlights at night.

8. 🌄 Noise Reduction (Again)

- Second noise reduction stage may be done after color processing to smooth chroma noise.

9. 💬 Color Space Conversion

- **Problem:** You can't always keep image in RGB format — video encoders and displays prefer **YUV** (or other formats).
- **Fix:** ISP converts the image to **YUV**, which separates brightness from color info.
- **Use case:** Feed to HDMI display, record to disk, stream via Ethernet, etc.

⌚ Real-Time Loop

All these steps are done in **real-time**, usually at **30 or 60 FPS**, and must be:

- **Fast**
- **Deterministic** (every frame processed exactly the same way)
- **Robust** (shouldn't fail even under bad lighting or sensor faults)

Cogent's software seems to provide:

- Reusable real-time image processing software
- Tuning tools to optimize image quality
- Support for hardware acceleration (e.g., via GPU, DSP, or custom ISP blocks)

Summary of the Full Pipeline

pgsql

```
Sensor (Raw Bayer) →
  Raw Pixel Fix →
  Lens Correction →
  Noise Reduction (Spatial/Temporal) →
  Demosaic →
  Auto Exposure →
  Auto White Balance →
  Color Correction →
  Noise Reduction →
  Color Space Conversion →
  RGB/YUV Image → Display or ML
```

Would you like a diagram or code-level example (e.g., V4L2 pipeline or GStreamer-based implementation)?



Would you like a diagram or code-level example (e.g., V4L2 pipeline
yes

Great! Let's now break down the **ISP pipeline** in **simple, detailed steps** — particularly in the context of **automotive cameras**, where ISP is sometimes moved from the camera to the **head unit** or **surround view module**.

Why Move ISP to the Head Unit?

Traditionally:

- A camera had its own **CMOS sensor** and a **dedicated ISP chip**.
- That ISP chip processed the raw image data into a usable video format.

But in automotive:

- **Head units** already have powerful processors.
 - So we **save cost and reduce hardware complexity** by doing all image processing (ISP) inside the head unit — not inside the camera module.
-

🔧 What is the ISP (Image Signal Processor) Pipeline?

It's a **step-by-step real-time software or hardware flow** that takes **raw sensor data** and turns it into a **clean, corrected video frame**.

Here's a breakdown of each ISP function, with analogies to make it intuitive:

📷 ISP Pipeline Steps (Simplified)

Step	What It Does	Analogy
1. Raw Data Correction	Fix bad pixels from sensor defects	Like skipping broken pixels in a damaged photo
2. Lens Correction	Correct warping (fisheye) and brightness unevenness	Like unbending a curved mirror image
3. Noise Reduction (1st pass)	Remove random colored dots (noise)	Like using a blur filter to clean grainy photos
4. Demosaicing	Convert one-color-per-pixel raw sensor data (Bayer) to full-color RGB	Like painting in missing colors on a black-and-white outline
5. Exposure Control	Adjust brightness to match light conditions	Like changing shutter speed in a photo
6. White Balance	Remove unnatural color tint (blueish/yellowish)	Like setting correct color temperature
7. Color Correction	Adjust color tones for different environments (sunlight, night, etc.)	Like applying photo filters that correct lighting
8. Noise Reduction (2nd pass)	Further clean image without losing detail	Like removing background buzz in audio editing
9. Color Space Conversion	Convert image from RGB to formats like YUV (used for compression/display)	Like converting file formats for compatibility

🧠 Why This Pipeline Matters in Automotive?

Because:

- Lighting changes **rapidly**: from tunnels to sunlight.
- **Safety** depends on clear visuals.
- Processing must happen in **real-time**.
- Image must be compatible with **ADAS, driver view, recording, and surround view** systems.

Implementation

- This pipeline can run in **software on a CPU, hardware-accelerated via GPU/DSP**, or using a **V4L2 (Video for Linux 2)** framework.
- Linux-based head units (like Android Automotive or custom Linux) often use **V4L2, GStreamer**, or **OpenCL/CUDA** for acceleration.

Would you like a **diagram** of this pipeline or a **code-level example** using V4L2?



V4L2 explain in simple words

Sure! Let me explain **V4L2 (Video4Linux2)** in **very simple words** — especially useful if you're working with **cameras** in Linux, like in automotive or embedded systems.

What is V4L2?

V4L2 (Video for Linux 2) is a **Linux kernel interface** that lets you **talk to video and camera devices**. Think of it as a **translator** between your camera hardware and your software app.

Why is V4L2 Important?

- When your app wants to **capture video, control the camera, or get frames** — it does this through V4L2.
- Works with **USB webcams, built-in laptop cameras, automotive cameras, ISP pipelines**, and more.
- It's like Linux's official way to say:
"Hey camera, give me a frame, and apply this setting!"

What Can You Do with V4L2?

 Action	 Description
Open camera	Connect to <code>/dev/video0</code> , <code>/dev/video1</code> , etc.
Set format	Set resolution (e.g., 1920x1080), pixel format (YUYV, MJPEG, etc.)
Start streaming	Begin the video stream (like turning on the camera)
Grab frames	Get video frames one-by-one

Action	Description
Apply controls	Set brightness, contrast, focus, exposure, etc.
Close	Stop streaming and disconnect

□ Typical Flow (Simplified)

```
c

// Pseudocode for using V4L2
1. fd = open("/dev/video0", O_RDWR);           // Open the camera
2. ioctl(fd, VIDIOC_QUERYCAP, &cap);          // Check capabilities
3. ioctl(fd, VIDIOC_S_FMT, &fmt);             // Set resolution and pixel format
4. ioctl(fd, VIDIOC_REQBUFS, &req);           // Ask for buffers
5. mmap(...);                                // Map buffers to app memory
6. ioctl(fd, VIDIOC_STREAMON, &type);         // Start streaming
7. loop: {
    ioctl(fd, VIDIOC_DQBUF, &buf);           // Dequeue a frame
    process(buf);                            // Do something with the frame
    ioctl(fd, VIDIOC_QBUF, &buf);             // Requeue buffer
}
8. ioctl(fd, VIDIOC_STREAMOFF, &type);        // Stop streaming
9. close(fd);                                // Close camera
```

🎯 Real-world Use

- Automotive systems use V4L2 to handle **front/rear/surround cameras**
- Android under-the-hood uses V4L2 in lower HAL layers
- Raspberry Pi camera? Also uses V4L2!
- Useful in **C, C++, Python (via OpenCV), GStreamer**, etc.

🧩 V4L2 + ISP?

If you're moving the **ISP from the camera to the CPU**, V4L2 can also control the **ISP pipeline blocks**, such as:

- Demosaic
- Color correction
- White balance
- Crop and scale

These are exposed as **V4L2 sub-devices** or **media controller** pipelines.

Would you like a **working C/C++ example** using V4L2 to capture frames? Or a diagram of the V4L2 camera flow?



confusing

Let's explain **V4L2** again in the **simplest possible way**, step by step — imagine you are **talking to a camera on Linux**.

What is V4L2?

V4L2 (Video4Linux2) is a tool inside Linux that lets your software **talk to camera hardware**.

Think of it like a **remote control** for cameras.

What Can V4L2 Do?

Using V4L2, your software can:

- Connect to the camera
- Change camera settings (like brightness or resolution)
- Start video recording or streaming
- Get one frame at a time
- Stop the camera

Where Is the Camera?

In Linux, every camera shows up as a **file** — like:

```
bash
/dev/video0
/dev/video1
```

Your program opens this file to talk to the camera.

How Does It Work?

Here's a simple **step-by-step explanation**:

1. Open the camera

Think of this like switching on the camera.

2. Set the resolution and format

You tell the camera: "Give me 640×480 images in JPEG."

3. Ask for memory buffers

These are places to put the images.

4. Start streaming

The camera starts sending pictures.

5. Get each frame

Your app grabs one frame at a time and does something with it.

6. Stop streaming and close

Done. Camera off.

Real-Life Analogy

Imagine you're a security guard:

- You have a **camera feed** on your screen.
- You **tune the screen** (resolution, zoom).
- You **press record** to start.
- You **watch each frame**.
- Then you **turn off the monitor** when you're done.

That's **exactly what V4L2 does**, but with software.

Simple Summary

V4L2 does...	Like...
Open camera (<code>/dev/video0</code>)	Plugging in a webcam
Set settings	Choosing resolution (1080p, 720p...)
Start stream	Turning on live video
Get frame	Taking a screenshot
Close camera	Turning it off

Would you like a **real example in code** or a **visual diagram** to make this even easier?



V4L2 exist on android

Yes, **V4L2 exists on Android** — but it is used differently than on desktop Linux. Here's a **simple explanation**:

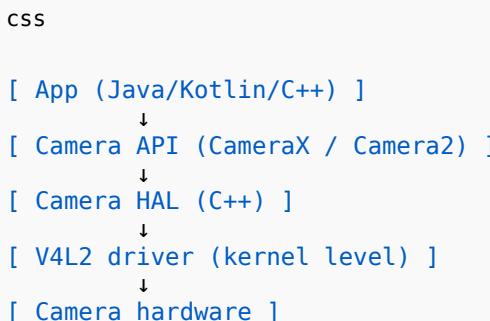
V4L2 on Android: Does It Exist?

Yes. Under the hood, Android uses **V4L2 (Video4Linux2)** to communicate with camera and video devices.

But you **don't use V4L2 directly** in Android apps — the Android **Camera HAL** and **Camera framework** sit on top of it.

Where V4L2 fits in Android

Here's the camera software stack on Android:



So:

- V4L2 = **Linux driver** layer
- Camera HAL = **Middleware** that converts V4L2 into Android-friendly calls
- App = Just uses Camera2 API or CameraX

Example: Taking a Photo on Android

1. You open your **camera app**.
2. The app uses **Camera2 API**.
3. Camera HAL (written in C++) talks to **V4L2 driver**.
4. V4L2 gets image from **camera sensor**.

When You Might Use V4L2 on Android

You might directly use V4L2 on Android **only if**:

- You're building a **custom Android device** (like in cars or industrial use).
- You're writing or debugging the **Camera HAL**.
- You're using **external USB cameras** (like `/dev/video0`).
- You're doing **low-level development** (e.g., in AOSP or NDK layer).

Debug Tip

You can often find V4L2 devices on Android by running:

```
sh
adb shell ls /dev/video*
```

If you see `/dev/video0` — that's a V4L2 device.

You can even test it with:

```
sh
adb shell v4l2-ctl --all
```

(If `v4l2-ctl` exists — from `v4l-utils`)

Summary

Question	Answer
Is V4L2 on Android?	 Yes, in kernel level
Can apps use V4L2 directly?	 No, use Camera2/CameraX
Who uses V4L2?	Camera HAL, USB camera apps, hardware devs
Where is it?	<code>/dev/video*</code>

Would you like to see a **real HAL example using V4L2** or a **simple diagram** showing how it flows?



yes

Great! Let's walk through both:

✓ 1. Diagram: V4L2 in Android Camera Stack

Here's a **simple diagram** showing how V4L2 fits in the Android camera pipeline:



- Apps don't touch V4L2.
- HAL developers do.
- V4L2 is just the standard Linux interface for video devices.

✓ 2. Real Camera HAL Using V4L2 (Example)

Below is a simplified snippet from a **custom Camera HAL** in Android that uses **V4L2**:

```

cpp
// Open the V4L2 device (e.g. /dev/video0)
int fd = open("/dev/video0", O_RDWR);
if (fd < 0) {
    ALOGE("Failed to open V4L2 device");
    return -1;
}

// Set V4L2 format
struct v4l2_format fmt = {};
fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width = 1920;
fmt.fmt.pix.height = 1080;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
ioctl(fd, VIDIOC_S_FMT, &fmt);

// Request buffers from V4L2 driver
struct v4l2_requestbuffers req = {};
req.count = 4;

```

```

req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.memory = V4L2_MEMORY_MMAP;
ioctl(fd, VIDIOC_REQBUFS, &req);

// Queue/dequeue buffers and start streaming...

```

This is **low-level C++ code** used inside Android's **Camera HAL implementation**.

✓ 3. How to Build a Custom Camera HAL with V4L2 (Simple Steps)

If you are making a **custom Android device** (like in cars, drones, etc.), and your camera gives V4L2 output, then:

Step 1: Write HAL code in C++ that:

- Opens `/dev/video0`
- Sets resolution and pixel format
- Requests and queues buffers
- Reads frame data from V4L2

Step 2: Create a HAL interface file (e.g., `ICameraProvider.hal`) using AIDL or HIDL.

Step 3: Integrate it with the Android CameraService.

✓ Summary Table

Layer	Technology Used	Example
App	Camera2 / CameraX	<code>CameraDevice.open()</code>
Framework	<code>system_server</code>	Talks to HAL via binder
HAL	C++ + V4L2	<code>/dev/video0, ioctl()</code>
Driver	V4L2 kernel module	Handles actual sensor
Hardware	Sensor	Produces raw video frames

Would you like a **full working sample HAL using V4L2** for Android (like for a USB camera)?



why we need V4L2

Great question! Here's a **simple explanation**:

✓ Why We Need V4L2 in Android (or Linux in general)

V4L2 = Video4Linux2

V4L2 is a standard Linux interface (API) to talk to video devices like:

- Camera sensors
- USB webcams
- TV tuners

Imagine This:

If you're a Linux system and someone plugs in a camera...

 "How do I talk to it? What's the resolution? Format? Start stream?"

Without a standard interface, every camera would need a different driver and API. Chaos!

✓ What V4L2 Provides

Feature	What It Does
 /dev/video0	Exposes camera device like a file
 Resolution/PixFormat	Set width/height/pixel format
 Buffer Management	Request, queue, and dequeue buffers
 Start/Stop Stream	Control video stream on/off
 Control I/O	Adjust brightness, gain, exposure, etc.

In Android:

- **HAL developers** use V4L2 to get raw frames from sensors.
- **Android apps never use V4L2 directly** — they use **Camera2 API**.
- **Android's Camera HAL** acts as a **bridge** between **CameraService** and the V4L2 driver.

Why It's Needed

Reason	Why It Matters
✓ Standard	Works with <i>many</i> types of cameras
🔧 Low-level control	Lets HAL manage frames, controls, buffers directly
🧩 Pluggable	USB or MIPI cameras just work via <code>/dev/videoX</code>
📸 Performance	Efficient MMAP or DMA buffer streaming

🧠 Without V4L2?

Every vendor would need:

- Their own kernel driver
- Their own user-space API
- No common HAL interface
- Poor portability and app compatibility

✓ TL;DR

V4L2 gives Linux (and Android HAL) a standard way to talk to camera/video hardware — set formats, manage buffers, stream frames — so everything works smoothly across different hardware.

Would you like a **V4L2 driver block diagram** or **code to test V4L2 camera on Android/Linux?**

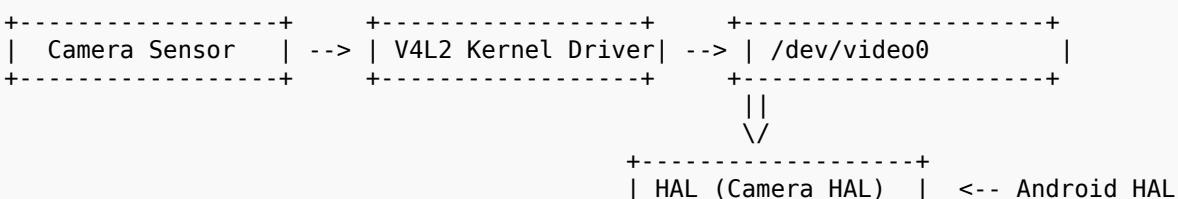


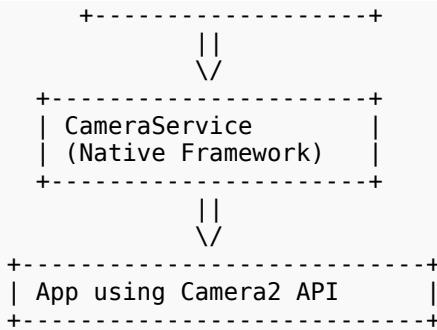
yes

Here's a **simple explanation with a block diagram** and **example code** to help you understand how **V4L2 works inside Android/Linux**.

✓ V4L2 Camera Driver Flow (Block Diagram)

plaintext





🧪 Example: Capture Frame using V4L2 in C (Linux)

This basic C example captures one frame from `/dev/video0`.

```
c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/videodev2.h>
#include <sys/ioctl.h>
#include <sys/mman.h>

int main() {
    int fd = open("/dev/video0", O_RDWR);
    if (fd < 0) {
        perror("Open failed");
        return 1;
    }

    // Set format: 640x480 YUYV
    struct v4l2_format fmt = {0};
    fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    fmt.fmt.pix.width = 640;
    fmt.fmt.pix.height = 480;
    fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
    fmt.fmt.pix.field = V4L2_FIELD_NONE;
    ioctl(fd, VIDIOC_S_FMT, &fmt);

    // Request buffer
    struct v4l2_requestbuffers req = {0};
    req.count = 1;
    req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    req.memory = V4L2_MEMORY_MMAP;
    ioctl(fd, VIDIOC_REQBUFS, &req);

    // Query buffer
    struct v4l2_buffer buf = {0};
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
    buf.index = 0;
    ioctl(fd, VIDIOC_QUERYBUF, &buf);

    // mmap
    void *buffer = mmap(NULL, buf.length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, buf.m.offset);

    // Queue buffer
}
```

```

ioctl(fd, VIDIOC_QBUF, &buf);

// Start streaming
enum v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ioctl(fd, VIDIOC_STREAMON, &type);

// Dequeue buffer (capture frame)
ioctl(fd, VIDIOC_DQBUF, &buf);

// Save frame
FILE *f = fopen("frame.yuyv", "wb");
fwrite(buffer, buf.bytesused, 1, f);
fclose(f);

// Cleanup
ioctl(fd, VIDIOC_STREAMOFF, &type);
munmap(buffer, buf.length);
close(fd);

printf("Frame saved to frame.yuyv\n");
return 0;
}

```

On Android (low-level testing via adb shell)

```

bash

adb shell
cd /dev
ls video*
# output: video0

v4l2-ctl --device=/dev/video0 --all
v4l2-ctl --device=/dev/video0 --stream-mmap --stream-count=1 --stream-
to=/data/local/tmp/frame.yuyv

```

You need v4l-utils:

Install on Linux:

```

bash

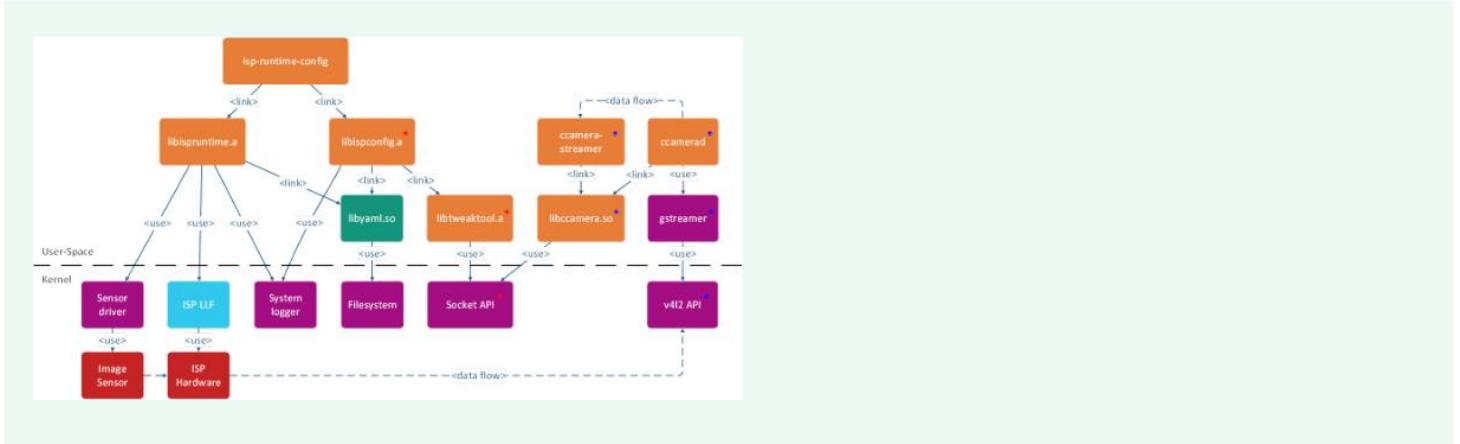
sudo apt install v4l-utils

```

Install on Android (with busybox or root):

- Push **v4l2-ctl** binary to device
- Use it from adb shell

Do you want a minimal **V4L2 HAL implementation** in Android or **V4L2 debug commands**?



Great! Let's break down this **Embedded ISP Architecture Diagram** into **simple, clear steps** to understand how the whole system works — from **image sensor** to **application streaming**.

🧠 What Are You Looking At?

This is a **hybrid embedded ISP system** using V4L2 and user-space tools to:

- Configure and control an image sensor and ISP
- Stream video
- Allow flexibility using config files and modular libraries

⌚ Flow Overview (Step-by-Step)

1. Kernel Space (Bottom)

Where hardware drivers live.

- **Image Sensor:** Captures raw images.
- **ISP Hardware:** Image Signal Processor — improves image quality (noise reduction, white balance, etc.).
- **Sensor Driver:** Talks to the sensor.
- **ISP LLF:** Low-Level Firmware interface to control ISP.
- **v4l2 API:** Exposes camera as `/dev/video0`, `/dev/video1`, etc. via kernel.

2. User Space (Above Dashed Line)

Where applications, libraries, and configs exist.

🔧 Control Layer

- **libispruntime.a**
 - Core control library for running the ISP in real time.
 - Uses:
 - Sensor drivers
 - ISP firmware
 - Logging
 - File system
- **isp-runtime-config**
 - Contains AE (Auto Exposure), AWB (Auto White Balance), AGC (Auto Gain Control) logic configs in YAML.
 - Linked into `libispruntime.a`.

📁 Config & Tools

- **libispconfig.a**
 - Reads/parses ISP config files.
- **libyaml.so**
 - Reads `.yaml` files (sensor tuning, runtime config).
- **libtweaktool.a**
 - Optional library to fine-tune ISP settings dynamically via socket.
- **System logger**
 - Collects ISP/system logs.

3. Camera Streaming Stack

Where the image data flows to apps.

- **ccamerad**
 - Camera daemon: Initializes and controls V4L2 + ISP, manages sensors.
 - Uses **gstreamer** and **V4L2 API** to fetch processed frames from hardware.
- **ccamera-streamer**
 - App-level streamer to push frames over network or to app.
- **libccamera.so**

- Shared library to interact with camera functions (load config, start stream, etc.).
- **Socket API**
 - Optional: For tuning/debugging/remote commands (via `libtweaktool.a`).

Data Flow

- **Image data:** From Sensor → ISP → V4L2 API → `gstreamer/libcamera.so` → App
- **Config/Control:** From `.yaml` files → parsed by `libyaml.so` → used by `libispruntime.a` and others
- **Commands/logs:** Go via sockets, logs, or file system

Why This Design?

Advantage	Explanation
Modular	You can update tuning/config without recompiling the whole system.
Portable	Can be adapted across hardware platforms.
No ISP chip needed	ISP is handled in existing SoC.
V4L2-compatible	Standard Linux camera interface.
Supports streaming + tuning	Via sockets or streaming libraries like GStreamer.

Would you like me to annotate the diagram or show how this ties into Android or Linux apps?



PC Tuning Tools

Every professional needs good tools. This is certainly true when it comes to developing an image pipeline: The large number of different settings that can be adjusted and their interdependence requires a sophisticated set of tools to get the job done. At Cogent, we have developed our own suite of ISP tuning tools. Based on camera profile data we can quickly provide an initial setting, and optimize those settings within a closed loop process: The PC “watches” the camera output and manipulates the embedded ISP on the fly through our proprietary Tweaktool API.

This paragraph explains how *PC Tuning Tools* are used to **configure and optimize the ISP (Image Signal Processor)** in embedded camera systems, such as those in vehicles. Here's a breakdown in **simple words**:

What are PC Tuning Tools?

They are special **software tools** run on a computer (PC) to help camera developers **fine-tune image quality** (sharpness, brightness, color, etc.) coming from the camera.

Why are they needed?

- Cameras have **many adjustable settings** (exposure, white balance, noise reduction, etc.).
 - These settings are often **connected to each other** — changing one may affect the other.
 - It's **very hard to tune** everything manually or blindly.
 - A **good tool helps do this faster and more accurately**.
-

How does Cogent's PC tuning work?

1.  Cogent has built its own **custom tuning software** for ISP.
 2.  The PC **monitors the camera output in real-time**.
 3.  It uses the **Tweaktool API** to **change the camera ISP settings on the fly**.
 4.  This creates a **closed-loop system**:
 - The PC looks at the image.
 - Adjusts ISP settings automatically or manually.
 - Checks if the image quality improved.
 - Repeats until the image looks good.
-

Summary

PC tuning tools are like smart remote controls for the camera image processor. They help engineers quickly find the right balance of image settings to make the picture look great — even in tough lighting or noisy environments — without needing to change the firmware over and over again.

Let me know if you want to see what typical ISP tuning options look like or how they relate to V4L2 or Android.



V4L2 is api interface?

Yes, **V4L2 (Video4Linux2)** is an **API interface** — specifically a **Linux kernel API** — that allows software (like camera apps, media players, or ISP tuning tools) to **communicate with video devices** such as:

- USB webcams
- CSI cameras (e.g., in Raspberry Pi or Android devices)
- TV tuners
- Video capture cards
- ISP hardware blocks

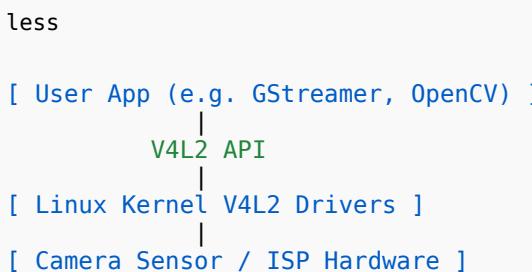
In Simple Terms:

V4L2 is like a translator between your software and your camera hardware.

It provides **standard functions** that apps can use to do things like:

- Open a camera
- Start/stop streaming video
- Set camera parameters (brightness, white balance, resolution)
- Grab frames from the camera

Where V4L2 fits:



Examples of What V4L2 Does:

- **VIDIOC_S_FMT** → Set image format like YUV or RGB
- **VIDIOC_REQBUFS** → Allocate memory buffers
- **VIDIOC_QBUF / VIDIOC_DQBUF** → Queue and dequeue image frames
- **VIDIOC_STREAMON / STREAMOFF** → Start and stop the video stream

Why It's Useful:

- It gives a **standard interface** no matter what camera hardware you use.
- Linux apps can work with many camera types **without custom drivers**.

- It allows **kernel-mode camera drivers** and **user-mode applications** to work together safely.
-

Let me know if you want a real code example of how to use V4L2 in C or how Android uses it under the hood.