V4L2 Stateless Video Decoder

How H.264 frame-based decoding works, explained with GStreamer v4l2slh264dec

Media Request API · DPB management · reference_ts · buffer lifetime

```
filesrc ! qtdemux ! h264parse ! v4l2slh264dec ! sink
```

SungHo(Jackson) Lee

Chips&Media

About Me

SungHo(Jackson) Lee

Software Engineer
Chips&Media

1

VPU kernel drivers & GStreamer
Develop and maintain the Wave5 V4L2 driver (drivers/media/platform/chips-media) — upstream Linux kernel work, plus stateful/stateless codec integration: V4L2 M2M, Media Request API, GStreamer decoder elements

2

Media framework development
Design and develop multimedia frameworks for Chips&Media hardware video codecs

Agenda

1. **Concepts**

   Stateful vs stateless, who does what, the Media Request API

2. **Architecture**

   The 3 GStreamer layers, h264parse and AU alignment

3. **Decoding flow**

   Negotiation, callbacks, filling parameters, submit, output

4. **Key mechanisms**

   DPB bumping, reference buffer lifetime, testing with visl

Stateful vs Stateless Decoders

Both are V4L2 M2M devices. The difference: who keeps the state?

Question	Stateful	Stateless
Who parses the bitstream?	driver / firmware	userspace
Who keeps SPS / PPS and the DPB?	hardware	userspace
What does userspace send?	compressed data only	data + ALL parameters, every frame
Who decides the output order?	driver	userspace (DPB)
Hardware complexity	high (needs firmware)	low (pure execution engine)
Examples	`wave5` · `v4l2h264dec`	`rkvdec, visl` · `v4l2slh264dec`

Key idea: a stateless driver works like a pure function — every frame arrives with a complete set of inputs

Who Does What: Userspace Is the Brain, the Driver Is the Hands

Userspace (GStreamer)

1. **Parse NALs** — find NAL boundaries (start codes / length fields)
2. **Manage SPS / PPS** — store them, pick the active ones
3. **Compute POC** — the number that sets the display order
4. **Manage the DPB** — track references, bumping, MMCO / sliding window
5. **Point at references** — dpb[16].reference_ts names the capture buffer

Kernel Driver

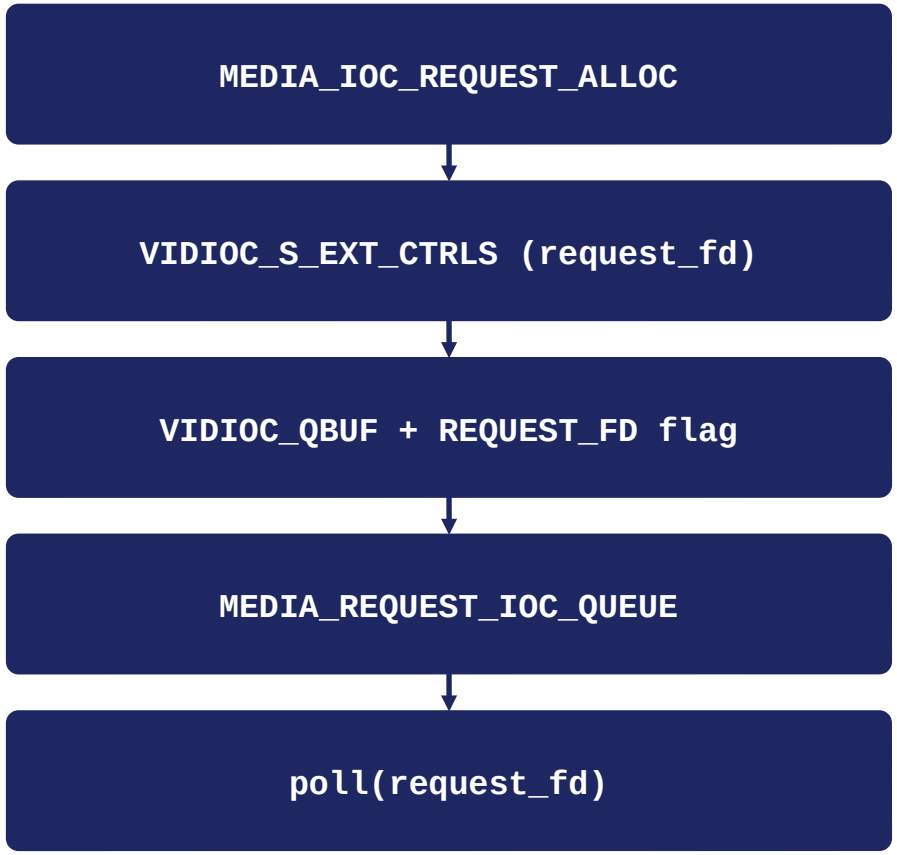
- control values → HW registers
- buffer queues (vb2)
- kick decode + report done
- no memory between frames

controls + buffers = a complete input, every frame

The Key Tool: Media Request API

Buffers and controls must travel together, as one package

Problem: normal S_EXT_CTRLS is global. With many frames in the queue, the driver cannot tell which SPS/DPB belongs to which frame.



get a request fd from the media device (REINIT to reuse)

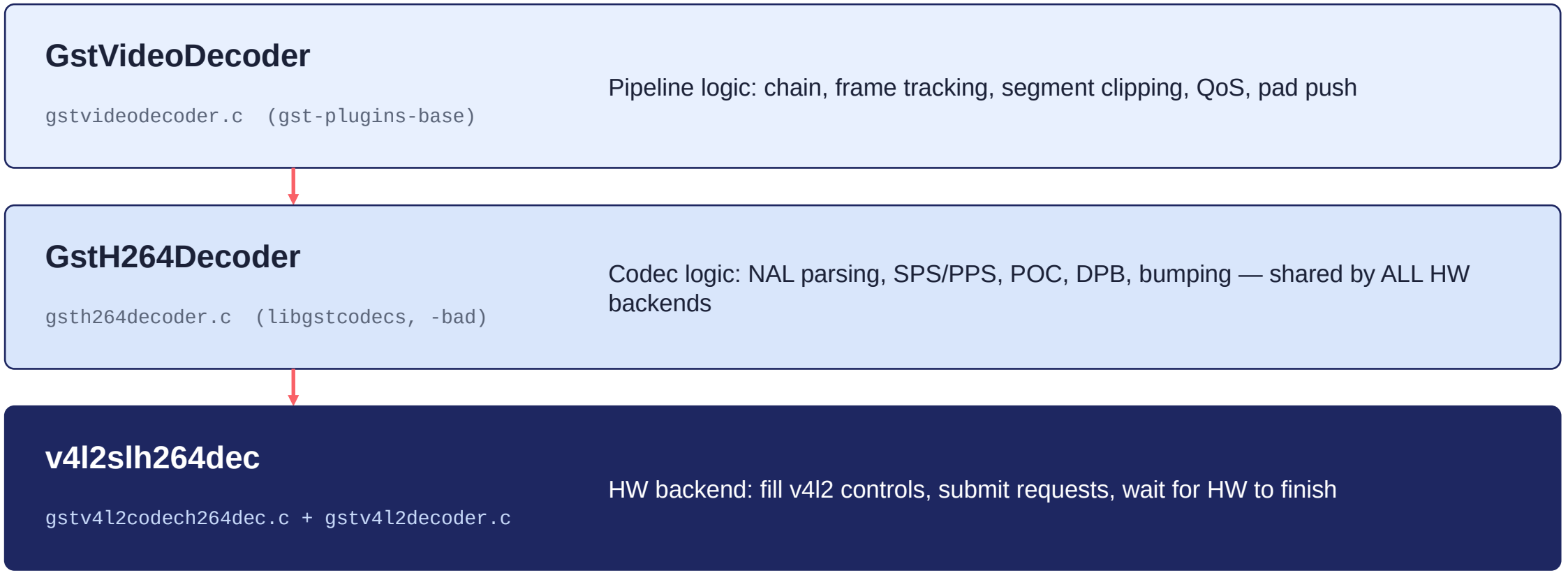
attach SPS, PPS, DECODE_PARAMS to the request

attach the OUTPUT (bitstream) buffer to the request

the whole package goes to the driver at once → decode starts

wait for done → collect buffers with DQBUF

GStreamer Architecture: 3 Clean Layers



vfunc borders: handle_frame (top→middle) · new_sequence / start_picture / decode_slice / end_picture / output_picture (middle→bottom)

Precondition: h264parse and AU Alignment

The decoder never looks for frame boundaries by itself

h264parse's job — scan start codes → find NALs → find frame (AU) boundaries → pack one frame's NALs into one buffer

The decoder's rule — packetized mode: "1 input buffer = 1 picture (AU)", end of buffer = end of picture

Two stream formats — byte-stream (TS/RTP: start codes) vs avc (MP4: length fields) — after parsing, both become plain NALs

```
filesrc ! v4l2slh264dec  →  fails (not-negotiated).  h264parse is required.
```

Getting Started: Find the Device, Ask Its Mode

1

Plugin load

Find the /dev/mediaN + /dev/videoN pair (media controller is required).
Try S_FMT(V4L2_PIX_FMT_H264_SLICE) on the OUTPUT queue — the _SLICE format marks a stateless driver.

2

Element open

Read two read-only controls from the driver, and simply follow them:

`V4L2_CID_STATELESS_H264_DECODE_MODE`	FRAME_BASED / SLICE_BASED — the driver decides, userspace follows
`V4L2_CID_STATELESS_H264_START_CODE`	NONE / ANNEX_B — does the HW want a 00 00 01 start code?

Frame-based → set_process_ref_pic_lists(FALSE): no per-slice reference list reordering is asked from the base class

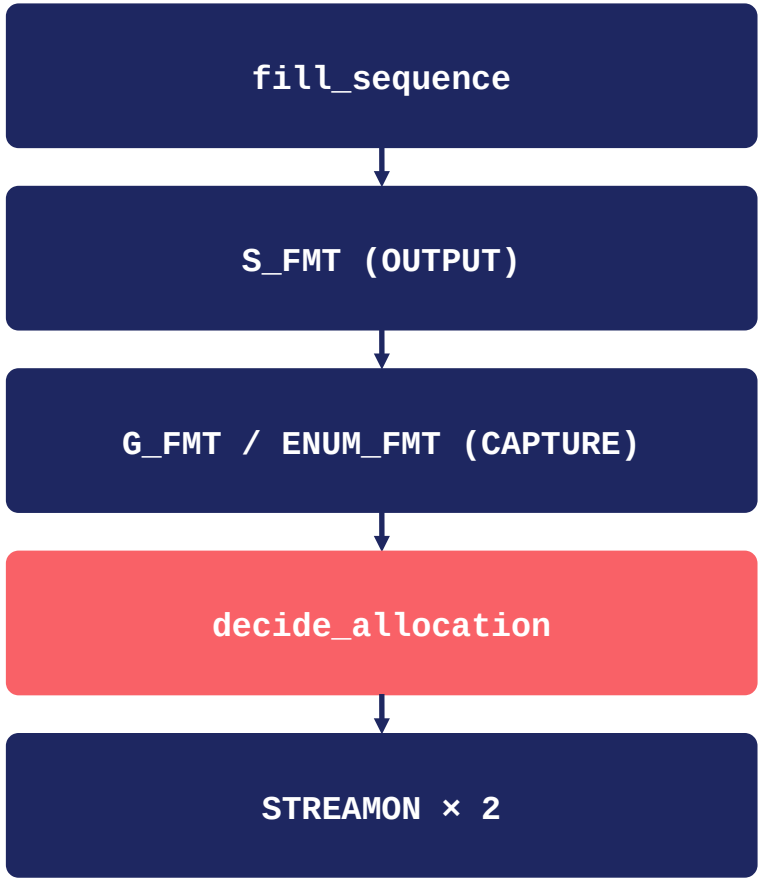
Frame-Based vs Slice-Based

Item	Frame-based	Slice-based
Request unit	1 per frame	1 per slice (sub-request)
OUTPUT buffer holds	all slices of the frame	one slice
SLICE_PARAMS / PRED_WEIGHTS	not used — driver parses slice headers itself	required for every slice
ref_pic_list0/1 reordering	done by the driver	computed by userspace
DECODE_PARAMS (with DPB)	required every frame	required every frame (same!)

The common core: in BOTH modes, the DPB and reference pointing are userspace's job — the modes only differ in granularity

new_sequence: Negotiation and Setup

Runs when the first SPS arrives, or when it changes (resolution, DPB size, ...)



convert the SPS into struct v4l2_ctrl_h264_sps and keep it

set H264_SLICE with the real size + set the SPS control first

the driver picks the output format (NV12, 10-bit → P010, ...) → negotiate caps downstream

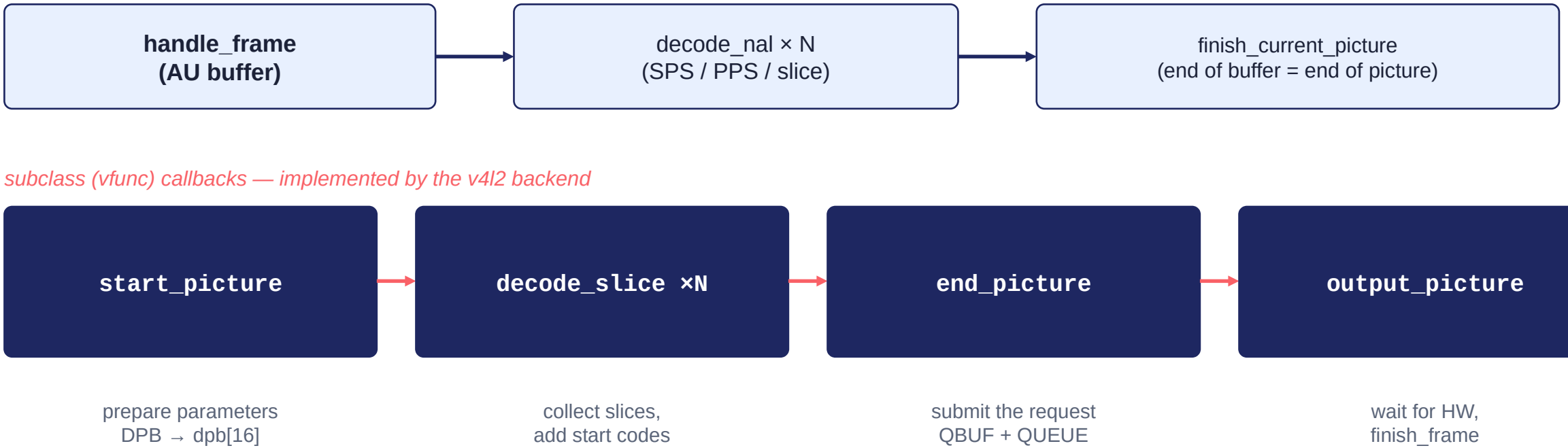
VIDIOC_CREATE_BUFS on CAPTURE — at least DPB size + some pipeline slack

start both the OUTPUT and CAPTURE queues

Too few buffers = deadlock: while references are alive, there is no free buffer left to decode new frames into

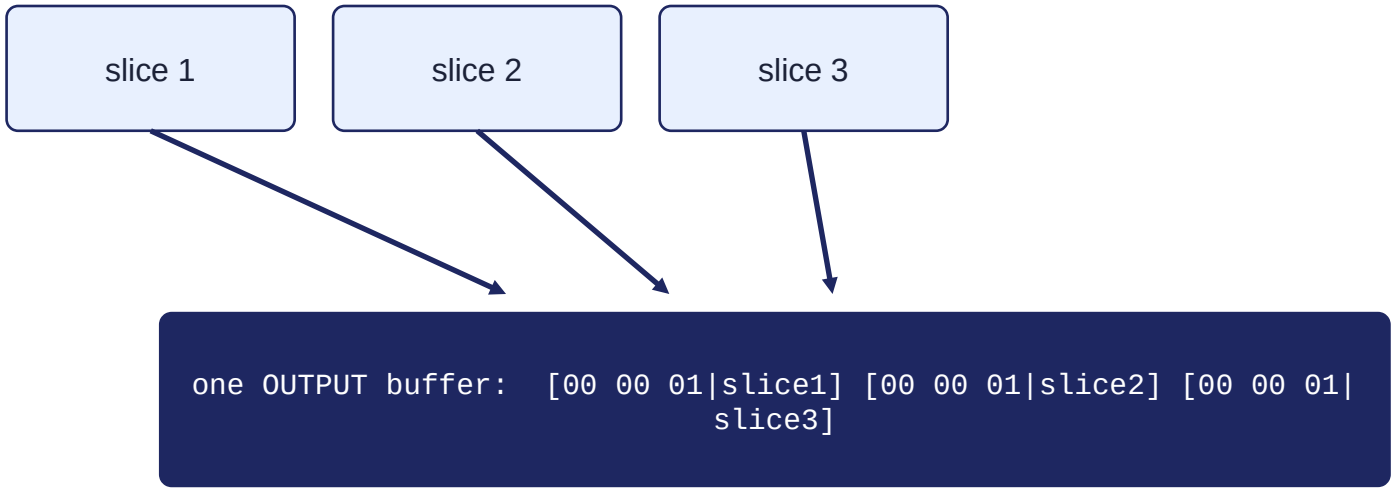
The Decoding Loop: Callback Chain

What happens when one AU buffer enters handle_frame



- SPS/PPS NAL → store it; if it changed → new_sequence (negotiation)
- First slice NAL → new picture is detected (POC computed, GstH264Picture created) → start_picture
- output_picture is NOT called from this loop — DPB bumping decides when (decode order ≠ display order)

decode_slice (Frame-Based): It Only Collects



(add a start code if the driver asked for ANNEX_B, then memcpy one after another)

the is_slice_based() block

- build SLICE_PARAMS — skipped
- build PRED_WEIGHTS — skipped
- reorder ref_pic_list0/1 — skipped

Why: the driver (HW / firmware) parses the slice headers itself — userspace only owns the parameter sets and the DPB

Even if the input came as AVC (MP4, length fields), it is parsed into NALs first, then rebuilt the way the driver wants (start codes)

end_picture → submit_bitstream: Send It

Once every slice of the frame is collected, submit in one go

1	`REQUEST_ALLOC / REINIT`	get a request fd (reused from a pool)
2	`S_EXT_CTRLS → request`	SPS/PPS only if changed; DECODE_PARAMS (DPB) every frame
3	`QBUF OUTPUT + REQUEST_FD`	bitstream buffer, timestamp = system_frame_number
4	`QBUF CAPTURE (plain)`	result buffer — not tied to the request, once per picture
5	`MEDIA_REQUEST_IOC_QUEUE`	the package is delivered atomically → HW starts decoding

Submitting is asynchronous — up to render_delay requests stay in flight; the oldest one is collected when the limit is hit

One Request, End to End: GStreamer → visl

What happens on each side of the kernel boundary

GStreamer (userspace) — makes the package

1. **Get a request fd**
   `MEDIA_IOC_REQUEST_ALLOC`
2. **Put controls in it**
   `S_EXT_CTRLS: SPS / PPS / DECODE_PARAMS`
3. **Put the bitstream in it**
   `QBUF OUTPUT + REQUEST_FD flag`
4. **Send the package**
   `MEDIA_REQUEST_IOC_QUEUE`

kernel boundary →

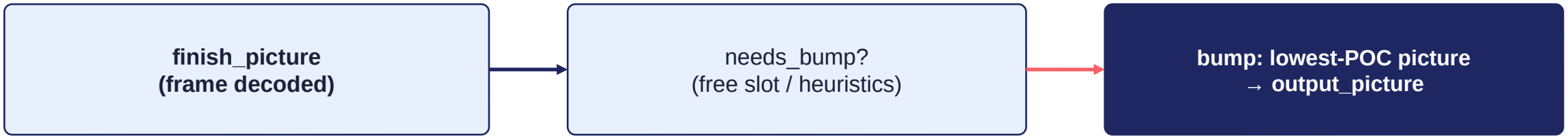
visl (kernel) — opens the package

1. **Check it** — `req_validate: exactly one buffer inside?`
2. **Queue the job** — `req_queue → v4l2_m2m_request_queue`
3. **Apply the controls** — `device_run: v4l2_ctrl_request_setup(src_req)`
4. **Read + "decode"** — `visl_find_control_data(SPS/PPS/DPB) → draw TPG`
5. **Finish** — `request_complete → buf_done (DONE)`

The point: request_setup makes the controls "current" only while THIS frame runs — that is how each frame gets exactly its own SPS/PPS/DPB

DPB Bumping: Deciding the Output Order (C.4.5.3)

Decode order ≠ display order, because of B-frames



Bump before insert — if the DPB is full, keep bumping until there is a free slot for the current picture (spec C.4.5)

Insert or output directly — reference or free slot → store in DPB. Non-ref + full DPB + lowest POC → skip the DPB, output right away

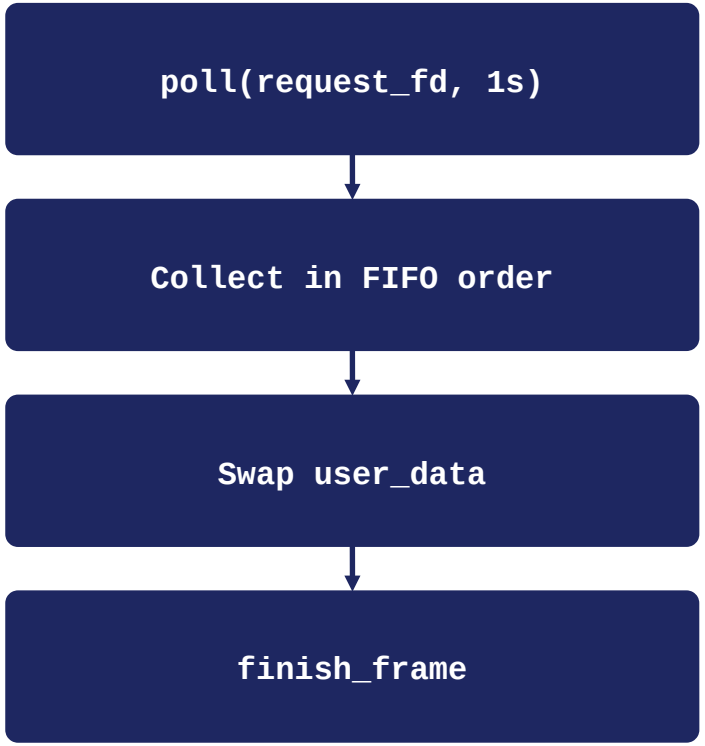
Low-latency mode — when is_live (LATENCY query): output early where the spec allows it (consecutive non-ref rule, POC gap ≤ 2)

Latency reporting — measure the REAL reorder depth with a decode-order counter → set_latency (no over-reporting)

Being output does NOT remove a picture from the DPB — reference pictures stay after they are shown

output_picture: Wait for HW, Then Push

Submits are async; the only waiting point is when a picture's display turn comes



wait for the HW to finish — timeout = error

M2M finishes in submit order → DQBUF everything up to this request (drop OUTPUT, check CAPTURE timestamp = frame#)

free the request; keep a CAPTURE buffer ref on the picture — so reference pixels stay valid while the DPB still needs them

remove from pending list → segment clipping / QoS → gst_pad_push(srcpad)

Pipelining throttle: keep at most render_delay requests in flight. Live = 0 (lowest latency) / VOD = N (HW never idles, best throughput)

Stateless Codec Support Today

Decoders: mature and in mainline · Encoders: still under development

Decoders — stable uAPI in mainline

H.264	`v4l2slh264dec`
HEVC	`v4l2slh265dec`
VP8 / VP9	`v4l2slvp8dec / v4l2slvp9dec`
AV1	`v4l2slav1dec`
MPEG-2	`v4l2slmpeg2dec`
FWHT	`test codec (visl / vicodec)`

Encoders — RFC stage, not in mainline yet

- No stable stateless encoder uAPI in mainline so far — patches are at the RFC / review stage
- RFCs exist: VP8 and H.264 encoding on Hantro H1 (tested on STM32MP25, with a GStreamer v4l2slh264enc branch)
- Design direction: userspace picks parameters and references; the kernel does rate control and bitstream generation
- A working group (Collabora, Pengutronix, Bootlin, ...) is driving the uAPI toward community consensus

Testing Tool: the visl Virtual Stateless Driver

drivers/media/test-drivers/visl — test the uAPI with no hardware at all

Fake decoding

no real decode — it draws debug info onto a test pattern (TPG) and returns that as the CAPTURE frame

Control tracing

dumps the submitted SPS/PPS/DECODE_PARAMS/dpb[] as ftrace events (trace_v4l2_ctrl_h264_*)

Reference checking

looks up dpb[i].reference_ts with vb2_find_buffer() and prints the result on the frame

Bitstream dumping

saves OUTPUT buffer contents through debugfs (bitstream_trace_* parameters)

Use it to: develop and test GStreamer / FFmpeg userspace code without hardware, catch bugs early while a new codec uAPI is still being designed, and compare traces against a known-good implementation

Inside visl: How the Driver Reads the Stateless Data

visl_device_run() — one frame, step by step (visl-dec.c)

1	**Pick the buffer pair**	`v4l2_m2m_next_src_buf() / next_dst_buf()`	the OUTPUT (bitstream) and CAPTURE buffers for this job
2	**Find the request**	`src_req = run.src->vb2_buf.req_obj.req`	the media request that was attached to the OUTPUT buffer
3	**Apply its controls**	`v4l2_ctrl_request_setup(src_req, &ctx->hdl)`	the SPS/PPS/DPB values from THIS request become the current control values
4	**Read the data**	`visl_find_control_data(V4L2_CID_STATELESS_H264_SPS, ...)`	v4l2_ctrl_find() → ctrl->p_cur.p — a plain pointer to the parsed structs
5	**Look up references**	`vb2_find_buffer(cap_q, dpb[i].reference_ts)`	turn each reference_ts into a real CAPTURE buffer (a HW driver would DMA from it)
6	**Finish the job**	`v4l2_ctrl_request_complete() → buf_done (DONE)`	controls are handed back to the request; both buffers are marked done

No parsing in the kernel: the data arrives already parsed as control payloads — the driver just reads structs and looks up buffers

Case Study: Stateless H.264 on Boda955

Our driver (coda-dec) — the same pattern as visl, driving real firmware

Every frame — device_run → start_process()

1. **Pick buffers + apply the request**
   `v4l2_m2m_next_src/dst_buf → v4l2_ctrl_request_setup(src_req)`

2. **Read the parsed data**
   `stateless_find_control_data(SPS / PPS / SCALING_MATRIX / DECODE_PARAMS) → p_cur.p`

3. **Feed the bitstream**
   `copy OUTPUT buffer into the ring buffer (BS_MODE_PIC_END), set rd/wr ptr`

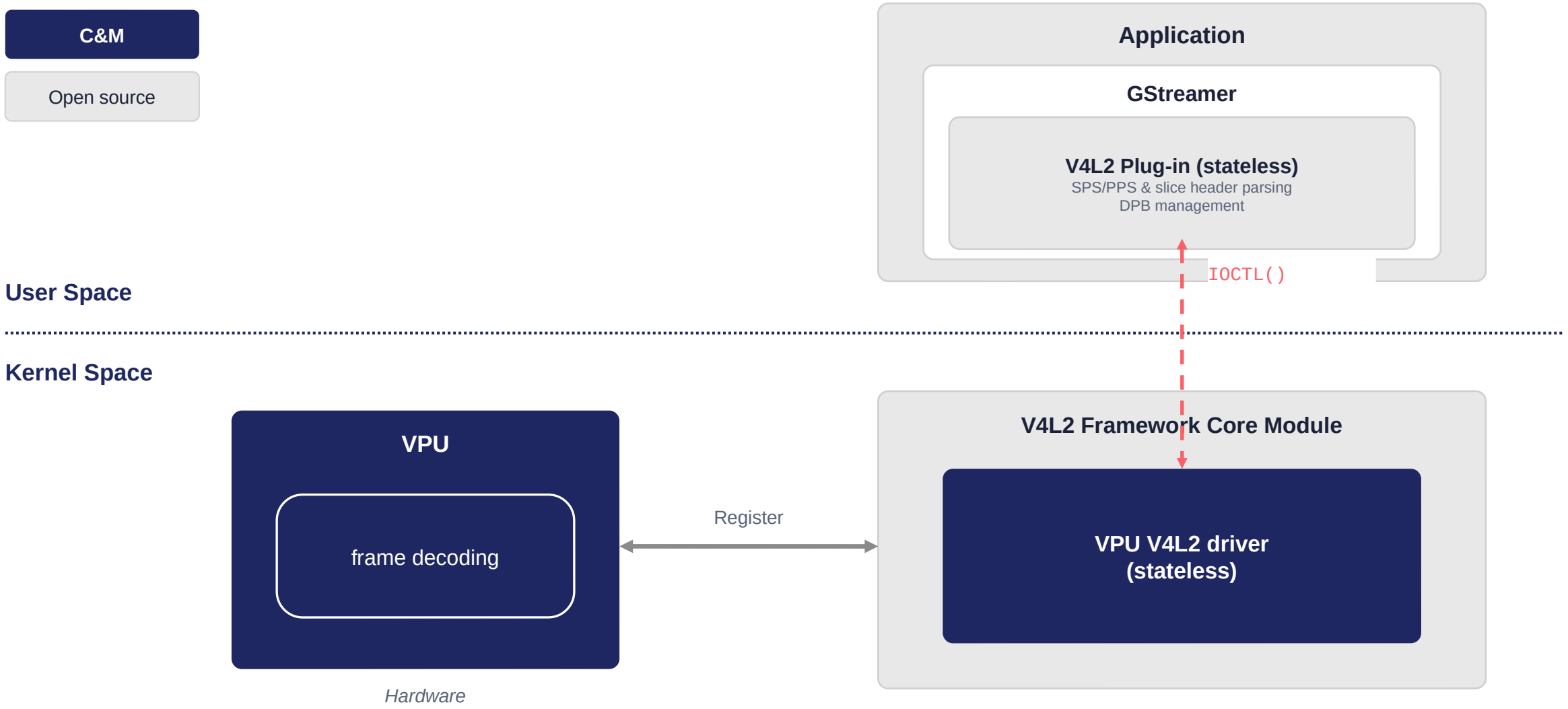
4. **Point at the output**
   `CAPTURE dma addr → CMD_DEC_PIC_STATELESS_LINEAR_ADDR_Y/CB/CR`

5. **Hand params to firmware**
   `SPS/PPS → avc_direct_memory struct → work buffer → PIC_RUN`

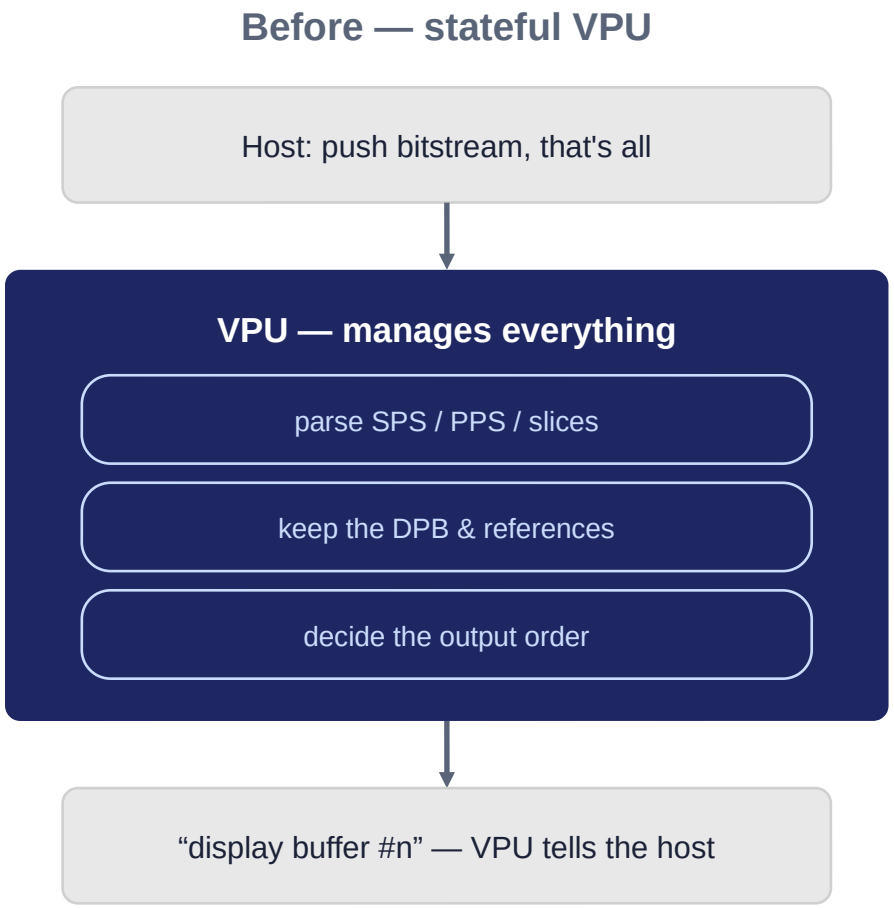
6. **Collect the result**
   `IRQ → RET_DEC_PIC_* / RET_DEC_STATELESS_LINEAR_ADDR_Y → buf_done (src + dst)`

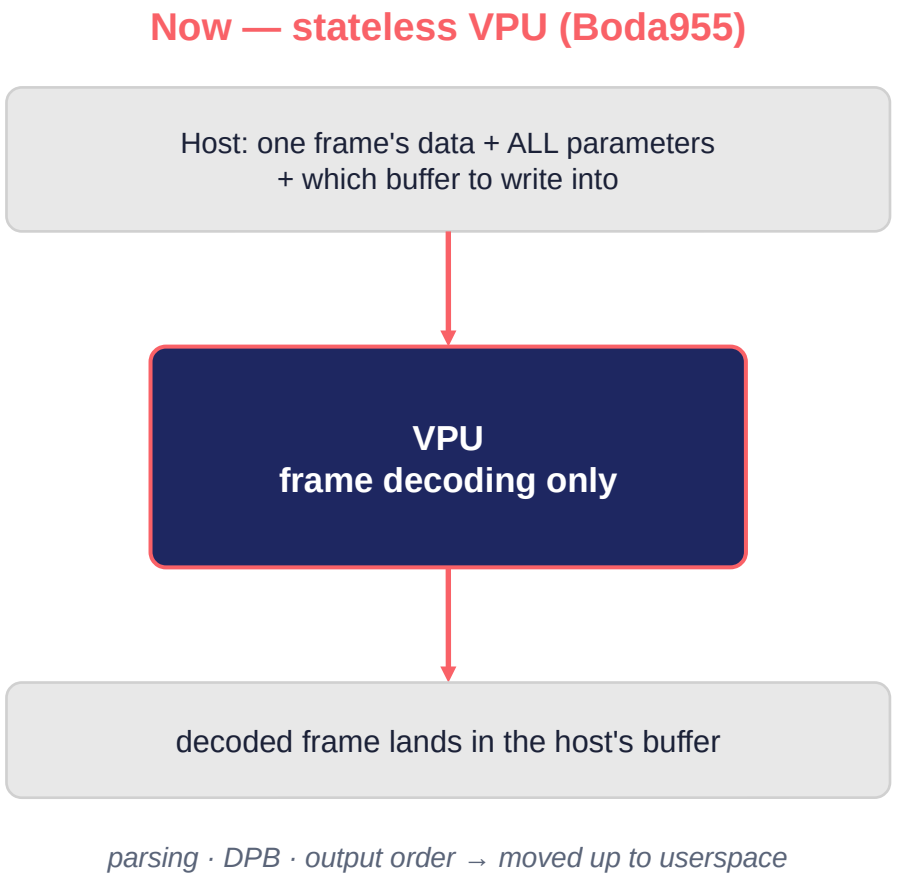
Where This Lives: Our VPU Software Stack

The stateless decode path, from GStreamer down to the VPU



The VPU Itself Had to Change





parsing · DPB · output order → moved up to userspace

The Same Idea Everywhere: VA-API and Vulkan Video

Stateless is not a V4L2 thing — it is how modern video APIs are designed

	V4L2 Stateless	VA-API	Vulkan Video
Target hardware	SoC fixed-function VPU (embedded)	desktop GPU (Intel / AMD)	any GPU, cross-platform
Who parses the bitstream	userspace	userspace	application
How parameters travel	controls + media request, per frame	VABuffer (picture params), per frame	VkVideoDecodeInfo + std headers, per frame
Reference management	app DPB + reference_ts	app picks VASurfaces	app manages DPB slots

Why it matters: all three follow the same stateless model — userspace parses, the driver executes. That is why GStreamer's shared codec layer (GstH264Decoder) can serve v4l2, VA-API, and Vulkan backends with the same parsing and DPB code.

Summary

1. Stateless = the state moves to userspace: parsing, POC, DPB, and reference tracking

2. The Media Request API delivers "buffer + controls" as one atomic package (1 request per frame in FB mode)

3. Output order comes from DPB bumping (lowest POC); buffer lifetime is protected by refcounts tied to the DPB

References

`kernel.org: userspace-api/media/v4l/dev-stateless-decoder · ext-ctrls-codec-stateless`
`GStreamer: gst-plugins-bad/sys/v4l2codecs, gst-libs/gst/codecs · kernel: drivers/media/test-drivers/visl`

Appendix: Multimedia Frameworks at Chips&Media

Which framework runs on which VPU — all validated with open-source tooling

Framework	VPU products	Standards & validation
VA-API	Boda955 · Wave5 · Wave6	H.264, HEVC, AV1, VP8/VP9, AVS2, H.263, MPEG-2/4, VC-1 · conformance on vaapi-fit with FFmpeg (GStreamer also runs)
V4L2 stateful	Coda9 · Wave5 · Wave6 · WaveJ	H.264, HEVC, MPEG-2/4, VP8, JPEG · v4l2-compliance + Fluster · stock GStreamer plugin, unmodified · Wave521C driver in mainline
V4L2 stateless	Boda955	H.264, MPEG-2, VP8 · v4l2-compliance + Fluster · stock GStreamer plugin, unmodified — this talk

Thank You

Questions?

SungHo(Jackson) Lee · Chips&Media