

Briefing Document: Linux GPU Driver Architecture & Direct Rendering Manager (DRM) Internals

1. Executive Summary

Modern Linux graphics drivers have evolved from legacy, unprivileged userspace direct hardware manipulation into a secure, object-oriented, transactional kernel subsystem governed by the **Direct Rendering Manager (DRM)**. To deliver high-performance 3D rendering, compute acceleration, and stutter-free display presentation, the subsystem enforces a strict division between user-space logic, kernel-space privilege, and hardware execution.

Key Takeaways

- **Architectural Separation (UMD vs. KMD):** User Mode Drivers (UMDs, e.g., Mesa 3D) compile high-level shading languages (GLSL/SPIR-V) into hardware microcode and construct command packets. Kernel Mode Drivers (KMDs, e.g., DRM drivers) act as privileged gatekeepers that manage memory address spaces, program display timing engines, dispatch command buffers to execution rings, and handle engine preemption and reset workflows.
- **Decoupled Device Node Security:** Functional capabilities are split across distinct character device nodes in `/dev/dri/`. Primary nodes (`cardX`) govern Kernel Mode Setting (KMS) and enforce single-master display ownership via `drm_master`. Render nodes (`renderD128+`) expose unprivileged, lockless GPGPU compute and 3D rendering capabilities to multi-tenant userspace applications, isolated by per-process GPU virtual address spaces.
- **Hardware Interface Foundations:** Low-level device interactions rely on PCIe Base Address Registers (BARs) and Memory-Mapped I/O (MMIO). Non-posted writes introduce synchronization hazards that require read-back pipeline flushes. Interrupt handling uses Multi-Vector Message Signaled Interrupts (MSI-X) split into a microsecond-bounded top-half Hard IRQ and a process-context bottom-half Threaded IRQ.
- **Transactional Atomic Modesetting (Atomic KMS):** Display configuration updates operate on an all-or-nothing, transactional model. Proposed display pipeline states across planes, CRTCs, encoders, and connectors are encapsulated in state objects and validated globally during an `.atomic_check` phase before touching hardware MMIO. Non-blocking commits defer hardware updates until upstream DMA fences signal completion.
- **Real-Time Display Pipeline & Synchronization:** Scanout DMA engines operate under hard real-time deadlines governed by pixel clocks and horizontal/vertical blanking intervals. High-precision presentation timestamps bypass CPU interrupt

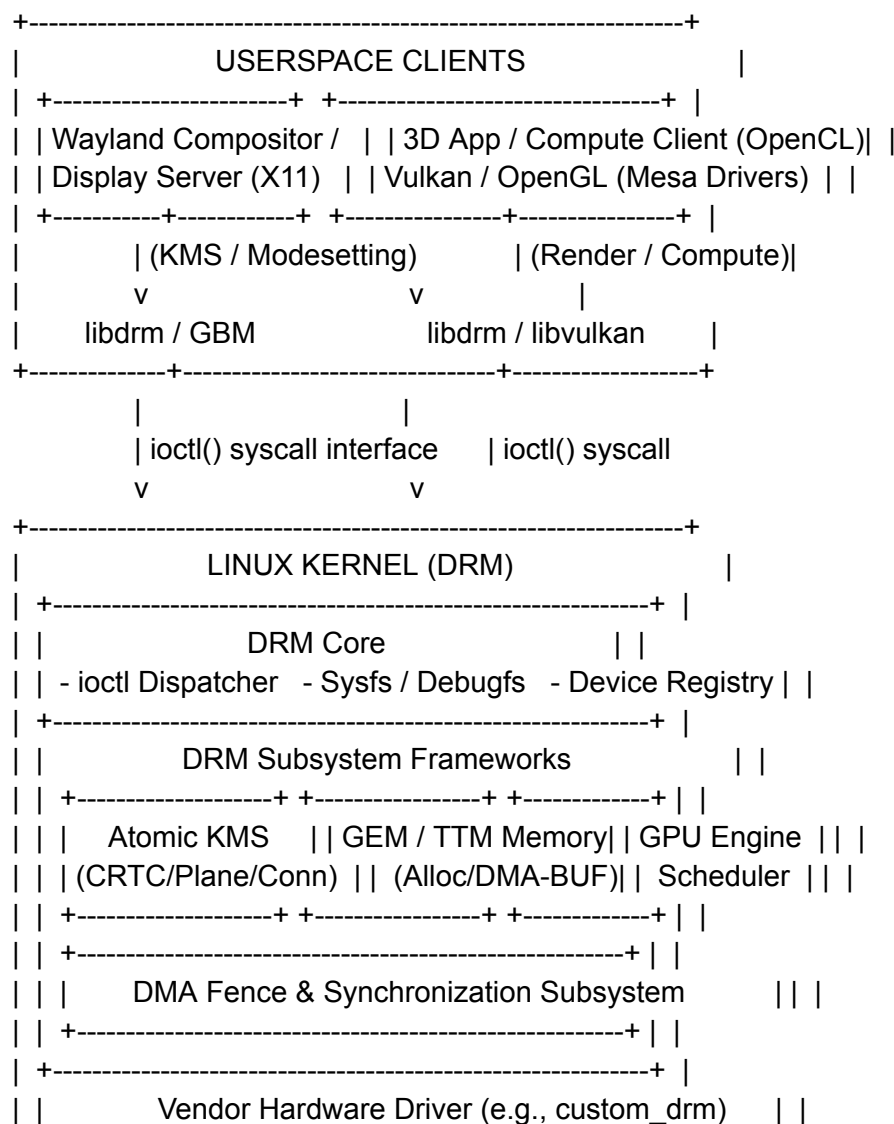
latency by reading real-time hardware raster scanline counters and projecting backward to the exact instant of frame presentation.

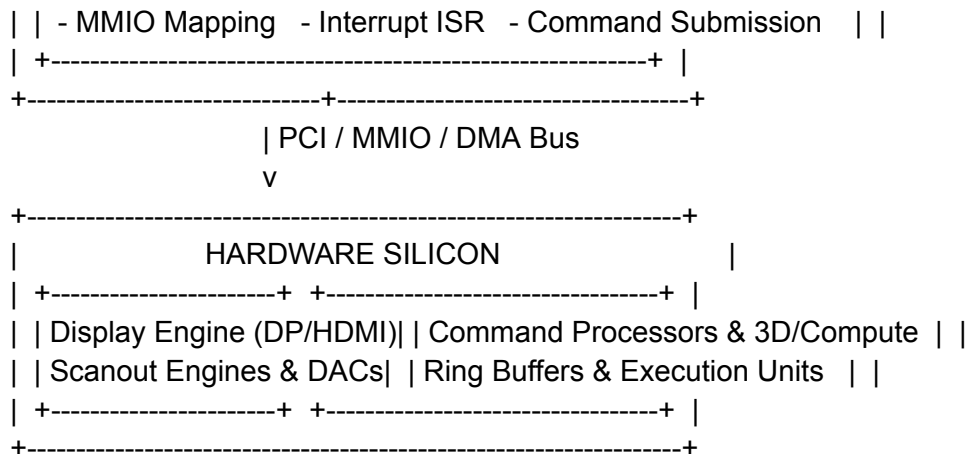
- **Unaccelerated Presentation via Dumb Buffers:** Unaccelerated display clients, boot loaders, and fallback compositors rely on the standardized Dumb Buffer API (`DRM_IOCTL_MODE_CREATE_DUMB`, `MAP_DUMB`, `DESTROY_DUMB`) to allocate scanout-ready GEM buffers backed by physically contiguous DMA-coherent memory or cached shmem pages.

2. Modern Linux Graphics Architecture & Subsystem Foundations

Evolution: Legacy User Mode Setting (UMS) vs. Modern DRM

Historically, Linux graphics relied on User Mode Setting (UMS), where the X Window System daemon executed with root privileges to directly manipulate I/O ports, map physical memory (`/dev/mem`), and program video clocks directly from userspace.





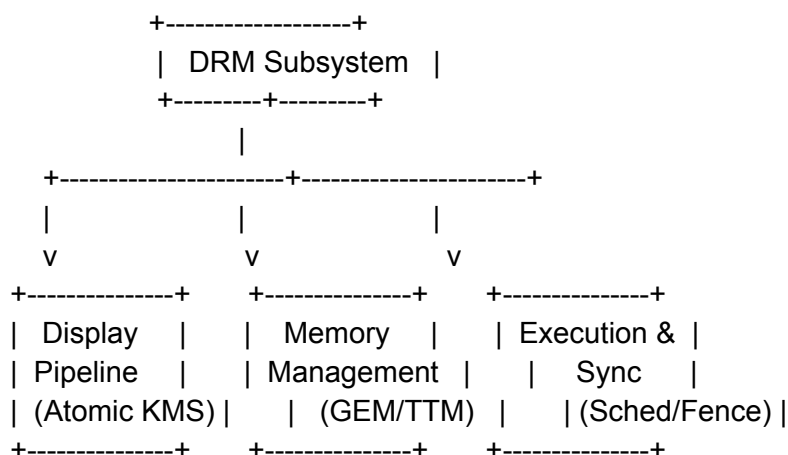
This legacy UMS architecture presented severe system stability and security risks:

- A crash in the userspace display server left the graphics hardware in an unrecoverable, frozen state.
- Virtual Terminal (VT) switching routinely triggered violent screen tearing, memory corruption, or hard system lockups.
- Multi-process access to the GPU lacked centralized kernel arbitration, allowing malicious or malformed user code to read arbitrary video memory.

Modern Linux graphics decouples execution policies, hardware control, and presentation primitives into a unified kernel-userspace stack governed by the Direct Rendering Manager (DRM).

The Fundamental Separation: UMD vs. KMD

The modern graphics stack is bisected across the kernel-user boundary into two complementary components:



1. **User Mode Driver (UMD):** Resides in userspace libraries such as Mesa 3D (RadeonSI, ANV, RADV, Iris, NVK) or proprietary compute runtimes. The UMD compiles high-level shading languages (GLSL, SPIR-V) into device-specific machine

microcode, builds indirect command buffers, tracks pipeline state objects, and constructs command packets.

2. **Kernel Mode Driver (KMD):** Implemented inside the kernel as a DRM driver module. The KMD acts as the privileged hardware gatekeeper. It does not parse high-level API logic; instead, it exposes low-level interfaces to map physical and virtual memory, programs display controllers, dispatches command streams to hardware execution ring buffers, manages Translation Lookaside Buffers (TLBs), and arbitrates preemption and hardware recovery.

The Three Core Pillars of DRM

- **Display Management (Atomic KMS):** Drives display pipelines by programming resolutions, color spaces, audio channels, and scanout timing generators through structured hardware abstractions (`drm_plane`, `drm_crtc`, `drm_encoder`, `drm_connector`, `drm_framebuffer`).
- **GPU Memory Management (GEM and TTM):** Allocates and tracks memory buffers. The **Graphics Execution Manager (GEM)** manages object lifecycles, user handles, and zero-copy sharing via DMA-BUF. The **Translation Table Manager (TTM)** manages complex video memory topologies in discrete GPUs featuring dedicated Video RAM (VRAM), enforcing memory eviction, handling system RAM bounce buffers, and driving GPU DMA migration engines.
- **Execution, Scheduling, and Synchronization:** Coordinates asynchronous GPU execution using the **DRM GPU Scheduler** (`drm_sched`) for job queuing and priority balancing, **DMA Fences** (`dma_fence`) for non-blocking inter-engine synchronization, and **DRM Sync Objects** (`drm_syncobj`) for timeline synchronization paradigms in Vulkan and modern graphics APIs.

Device Node Security & Access Matrix

The DRM core exposes hardware functionality to userspace via character device nodes located under `/dev/dri/` using major number 226:

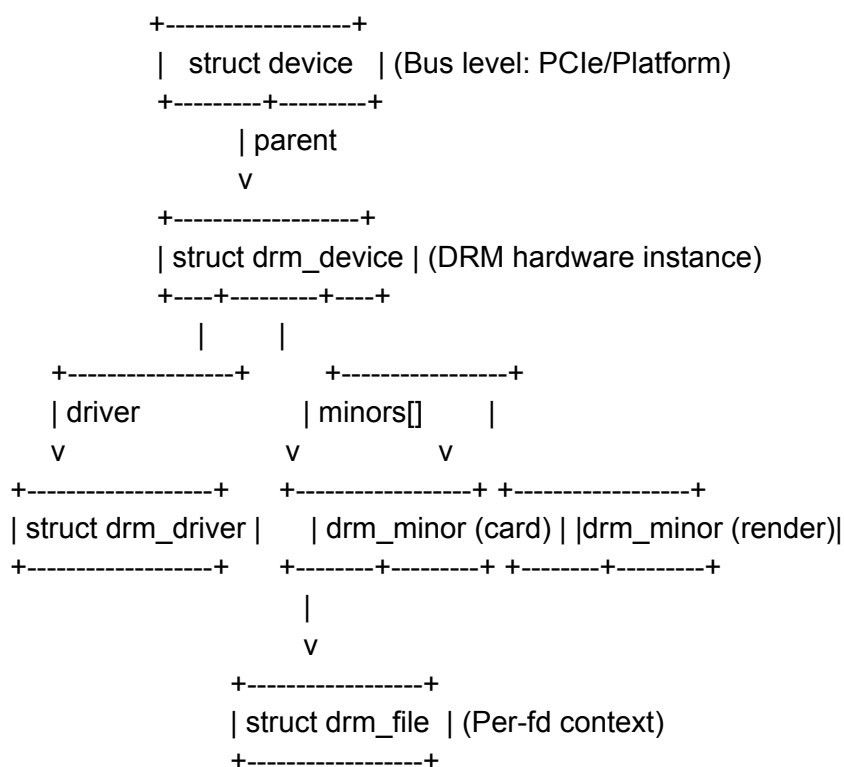
Device Node Path	Minor Range	Primary Subsystem Functions	Security & Access Permission Model
<code>/dev/dri/cardX</code>	0 to 63	KMS Modesetting, Display Control, GEM Handle Allocation	DRM Master / Authenticated Access. Only one process (e.g., Wayland Compositor) can hold Master status (<code>drm_master</code>) at a time to prevent display hijacking.

<code>/dev/dri/renderD128+</code>	128 to 191	Off-screen 3D Rendering, GPGPU Compute, Video Encode/Decode	Unprivileged Direct Access. Lockless execution model bypasses DRM Master checks entirely. Accessible by any client with standard Unix read/write permissions, isolated via per-process GPU virtual address spaces.
-----------------------------------	------------	---	---

3. Kernel Object Topography & Driver Lifecycle

Core Object Structural Hierarchy

The DRM core models graphics hardware through an object-oriented kernel graph:

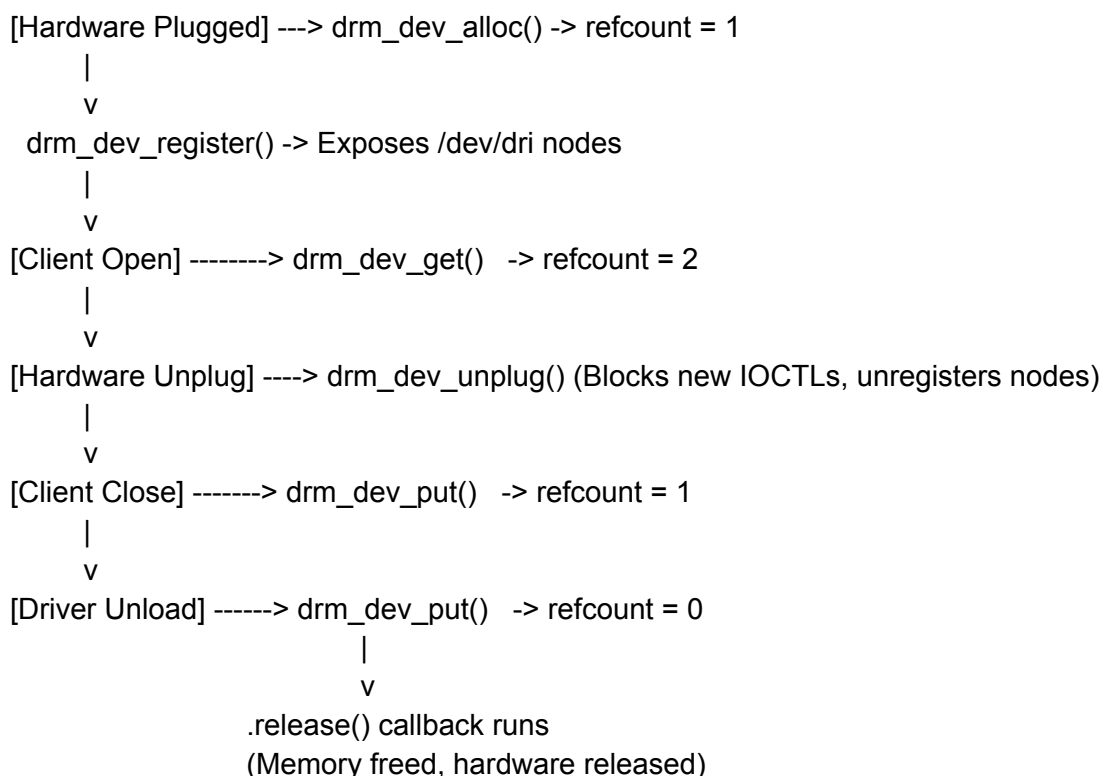


- `struct drm_device`: The central anchor representing a single logical GPU. It aggregates display configuration states (`struct drm_mode_config`), memory managers (GEM/TTM), lock primitives, registered client lists, and driver-private containers via `dev->dev_private`.
- `struct drm_driver`: The static dispatch table defining capabilities, feature flags (`DRIVER_GEM`, `DRIVER_MODESET`, `DRIVER_RENDER`, `DRIVER_ATOMIC`, `DRIVER_SYNCOBJ`), memory allocator hooks, and driver-private IOCTL declarations.

- `struct drm_minor`: Bridges the kernel device instance to filesystem character nodes (`/dev/dri/cardX` vs. `/dev/dri/renderD128`).
- `struct drm_file`: The per-client context instantiated whenever a process executes `open()` on a DRM device node. It maintains process-isolated handle tables (`object_idr`), master permissions, sync object timelines, and driver-private client tracking data.

Reference Counting & Safe Teardown

Because physical GPU hardware can be unbinded or hot-unplugged (e.g., Thunderbolt eGPUs) while userspace processes maintain open file descriptors or active memory mappings, DRM enforces a decoupled, two-tier teardown pipeline:

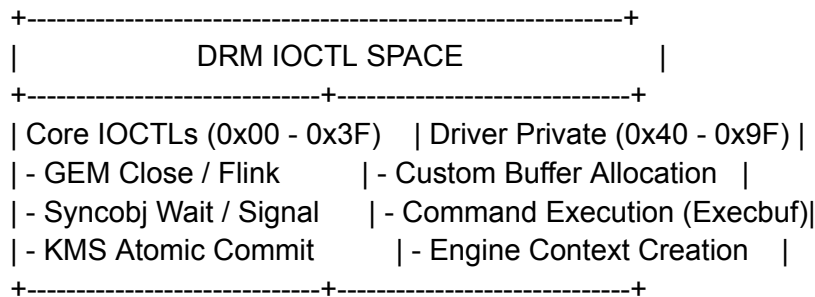


1. **Logical Detachment (`drm_dev_unplug`):** Flags the device as detached and immediately unlinks `/dev/dri/` filesystem character nodes. Ongoing system calls return `-ENODEV`, preventing new ioctls from reaching the hardware.
2. **Resource Reclamation (`drm_dev_put`):** The underlying memory for `struct drm_device` and driver-private allocations is retained until all internal reference counts (`dev->ref`) reach zero, guaranteeing that pending CPU memory mappings terminate safely without triggering kernel panics.

4. Userspace API Contracts & Capability Negotiation

Driver IOCTL Architecture & Dispatch

Userspace applications interact with a DRM driver using standard `ioctl()` system calls over file descriptors. The DRM core partitions the 32-bit IOCTL command space into two regions:



Driver-private IOCTLs are registered inside an array of `struct drm_ioctl_desc` containing bitwise execution flags:

- **DRM_AUTH**: Requires the calling client to be authenticated or hold Master status.
- **DRM_MASTER**: Restricts execution exclusively to the active DRM Master process (e.g., modeset operations).
- **DRM_ROOT_ONLY**: Restricts execution to processes possessing `CAP_SYS_ADMIN`.
- **DRM_RENDER_ALLOW**: Explicitly authorizes unprivileged execution through Render Nodes (`/dev/dri/renderD128+`).

Strict UAPI ABI Engineering Rules

Kernel-to-userspace Application Binary Interfaces (ABIs) cannot be modified once merged upstream. Designing driver-private UAPI structures requires strict rules to maintain compatibility across mixed 32-bit/64-bit execution environments (such as 32-bit x86 applications running on a 64-bit kernel):

1. **Explicit Fixed-Width Integer Types**: Use fixed-width types (`__u32`, `__u64`, `__s32`, `__s64`). Never use architecture-dependent C types such as `long`, `size_t`, or raw pointer types (`void *`).
2. **Explicit 64-Bit Alignment & Padding**: On 32-bit architectures, 64-bit integers align to 4-byte boundaries, whereas on 64-bit architectures, they align to 8-byte boundaries. All 64-bit members within UAPI structures must be aligned manually using explicit padding fields:

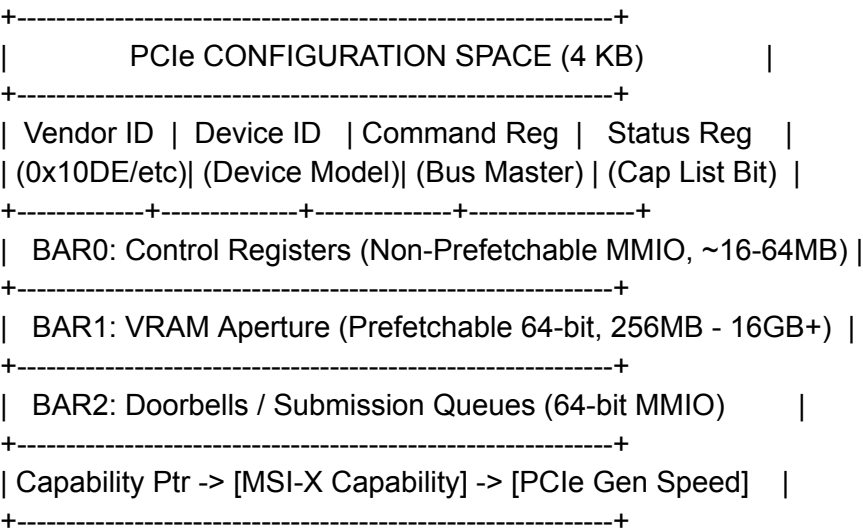
```
struct drm_custom_gem_alloc {
    __u64 size;    /* Offset 0, 8 bytes */
    __u32 domains; /* Offset 8, 4 bytes */
    __u32 handle;  /* Offset 12, 4 bytes */
    __u64 offset;  /* Offset 16, 8 bytes */
};
```

- 1. **Userspace Pointers as 64-Bit Integers:** Pass userspace virtual addresses as `__u64` fields cast from `uintptr_t`. In kernel space, convert them safely back to pointers using `u64_to_user_ptr()`.
- 2. **Zeroed Padding Enforcement:** Drivers must validate that all reserved padding bytes in incoming UAPI structures are strictly set to zero, returning `-EINVAL` if non-zero values are detected. This guarantees that reserved fields can be repurposed safely in future interface revisions.

5. Hardware Interface Foundations: PCIe, MMIO, & Interrupt Routing

PCIe Topology & Base Address Register (BAR) Roles

A discrete GPU exposes its hardware resources to the host operating system via its PCIe Type 0 Configuration Header:



Register Aperture	Memory Property	Typical Size Range	Functional Hardware Role	Host Caching Behavior
BAR0	Non-Prefetchable MMIO	16 MB – 64 MB	Control and Status Registers (CSRs), execution engine power control, interrupt mask tables, display timing controllers. Reads have immediate side effects.	Uncached (UC). Speculative reads forbidden.

BAR1	Prefetchable 64-bit Memory	256 MB – 24 GB+	Direct CPU access window to on-board Video RAM (VRAM). Supports PCIe Resizable BAR (ReBAR) extensions to map the entire frame buffer address space.	Write-Combining (WC). Speculative reads permitted.
BAR2 / BAR3	Prefetchable / MMIO	Low-latency pages	User-space and kernel submission doorbells for pushing queue pointers directly to silicon engines without global locking.	Uncached or Write-Combining .

Memory-Mapped I/O (MMIO) & Memory Ordering

MMIO maps physical GPU registers directly into the kernel's virtual address space via `ioremap()` or `pci_iomap()`.

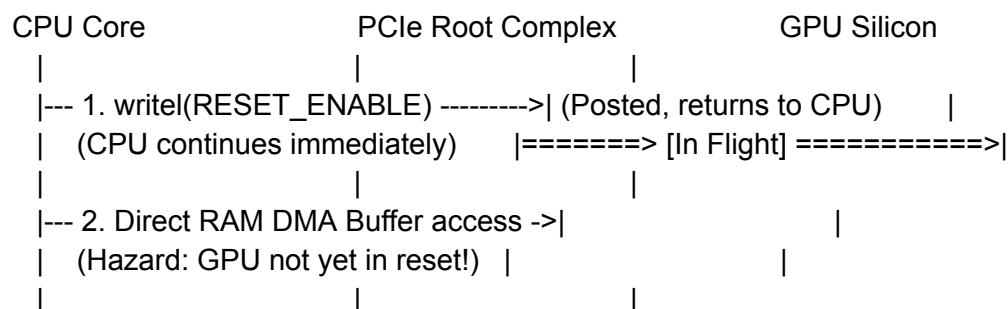
Register Access Primitives

Device registers must never be dereferenced through raw C pointers. Developers must use architecture-safe accessors (`readl()`, `writel()`, `ioread32()`, `iowrite32()`). Standard accessors insert implicit CPU bus memory barriers on weakly ordered architectures (ARM64, RISC-V). Relaxed accessors (`readl_relaxed()`, `writel_relaxed()`) omit memory barriers to minimize latency during bulk data transfers (such as firmware microcode ingestion), requiring explicit software barriers (`wmb()`, `rmb()`).

The Posted-Write Hazard and Pipeline Flushing

PCIe writes are **posted transactions**: the host CPU writes data to the PCIe root complex, which acknowledges the transfer immediately before the payload physically settles into the destination hardware register. PCIe reads are **non-posted transactions**: the CPU stalls until the completion packet returns from the PCIe endpoint.

This introduces an execution hazard where subsequent software steps might execute before a register write finishes propagating:



```
|--- 3. readl(RESET_STATUS) ----->| (Non-posted, blocks CPU)      |
|                                     |=====> [Flush Pipeline] =====>|
|<=====|===== [Data Returned] =====|
```

To force a posted write to complete immediately, the driver must execute a dummy read from the same MMIO aperture (`readl(base + REG_OFFSET)`), forcing the PCIe pipeline to flush all pending write operations.

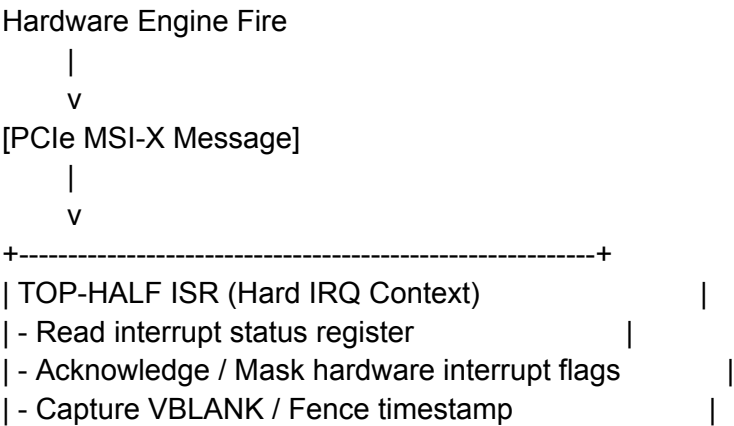
Interrupt Architecture: Legacy INTx vs. Multi-Vector MSI-X

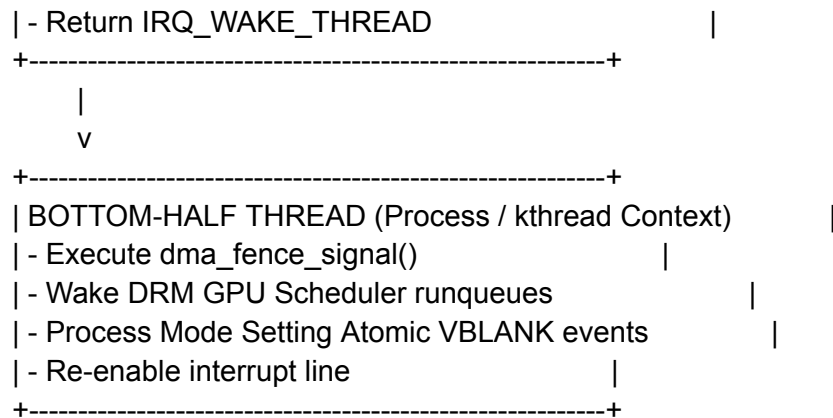
Modern GPUs produce massive interrupt streams across independent sub-engines. Handling these events efficiently requires advanced interrupt routing models:

Interrupt Model	Signal Transmission Mechanism	Vector Capacity	Core Affinity & Performance
Legacy PCI (INTx)	Physical sideband pin (INTA#–INTD#) shared across devices.	Single shared line.	High overhead. The CPU must poll every driver on the shared interrupt line to locate the source.
MSI-X (Extended Message Signaled)	In-band 32-bit DMA write to host interrupt controller (Local APIC / GIC).	Up to 2048 distinct vectors.	High efficiency. Independent vectors assigned per engine (e.g., Vector 0: Compute Ring; Vector 1: Display VBLANK), routed directly to specific CPU cores.

Top-Half vs. Bottom-Half Processing

To minimize time spent in hard interrupt contexts, processing is split into two phases:

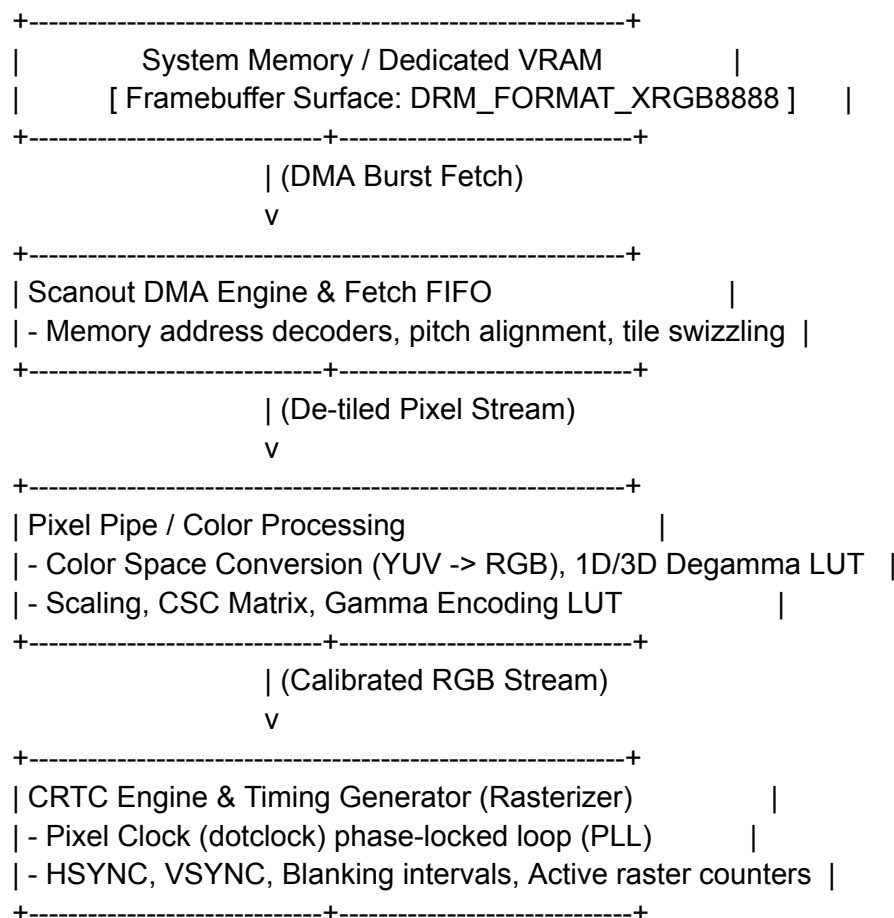


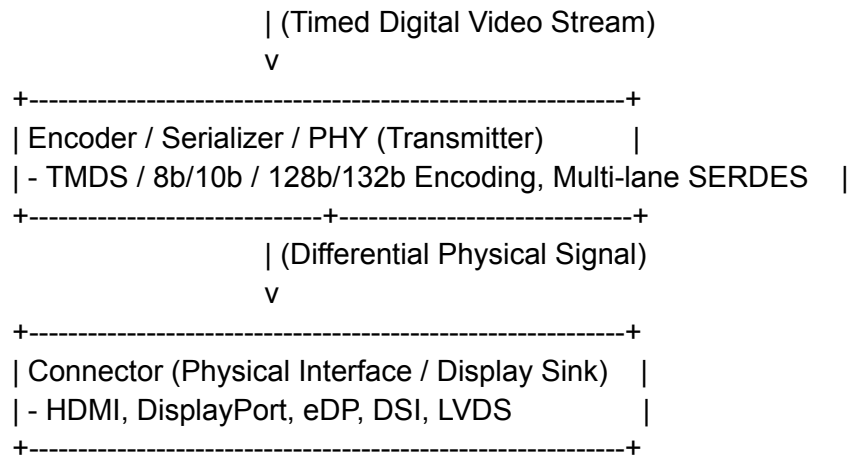


6. Display Pipeline Architecture & Kernel Mode Setting (KMS)

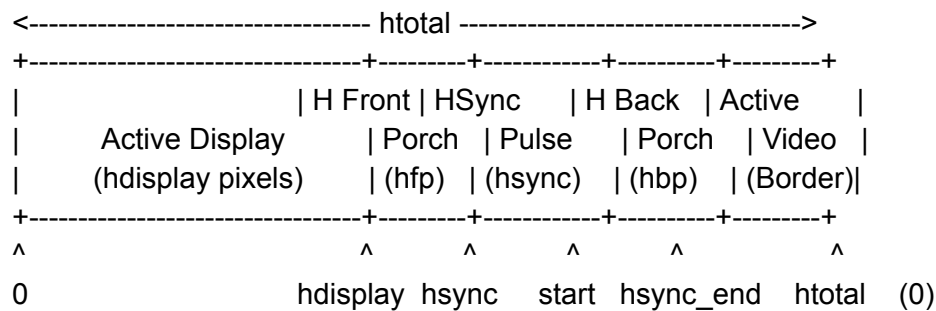
Real-Time Scanout Pipeline & Raster Mathematics

Unlike 3D rendering pipelines that operate asynchronously on command streams, a display engine is a real-time deterministic streaming engine. It must fetch pixel surfaces from memory, composite multiple layers, apply color transformations, serialize pixel streams, and drive high-speed PHY signals without stalling.





A display raster sweeps continuously across the screen in horizontal lines from top to bottom. The scanout timeline is partitioned into Active Video regions and Blanking Intervals:



Core Timing Definitions & Formulas

- **Active Area (**hdisplay**, **vdisplay**)**: Addressable visible pixel matrix (e.g., 1920 $\times$ 1080).
- **Front Porch (**hfp**, **vfp**)**: Delay interval between active video end and the leading edge of the sync pulse (**hsync_start** - **hdisplay**).
- **Sync Pulse (**hsync**, **vsync**)**: Pulse duration forcing display receiver horizontal line or vertical frame retrace (**hsync_end** - **hsync_start**).
- **Back Porch (**hbp**, **vbp**)**: Settling time before active scanout resumes (**htotal** - **hsync_end**).
- **Blanking Interval**: The sum of Front Porch + Sync Pulse + Back Porch.

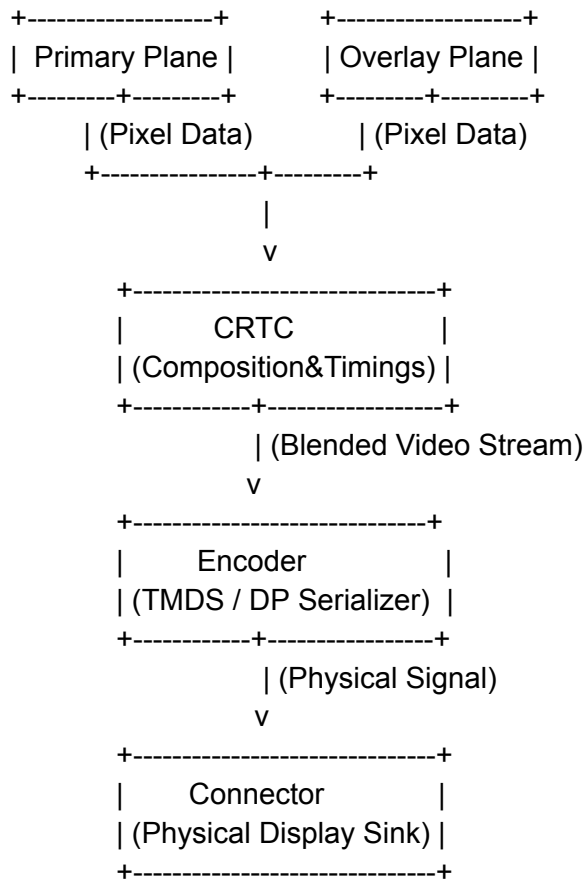
The pixel clock (dotclock in kHz) governs the rate at which pixels are emitted by the Phase-Locked Loop (PLL) timing generator:

$$\text{Pixel Clock (kHz)} = \frac{H_{\text{total}} \times V_{\text{total}} \times \text{Refresh Rate (Hz)}}{1000}$$

$$\text{Scanout Memory Bandwidth (B/s)} = \text{Pixel Clock (Hz)} \times \text{Bytes Per Pixel} \times \text{Active Planes}$$

The Four Core KMS Graph Entities

KMS models display hardware as a directed acyclic graph composed of four fundamental abstractions:



1. **Planes (`struct drm_plane`):** On-chip DMA fetch channels and scaling engines. They read pixel data directly from memory framebuffers (`struct drm_framebuffer`), perform format conversions, and apply positioning offsets. Categorized into Primary (`DRM_PLANE_TYPE_PRIMARY`), Overlay (`DRM_PLANE_TYPE_OVERLAY`), and Cursor (`DRM_PLANE_TYPE_CURSOR`) planes.
2. **CRTCs (`struct drm_crtc`):** Abstrate the hardware timing generators and composition mixers. A CRTC blends streams from assigned planes, applies global color LUT corrections, and drives the display raster. Generates VBLANK interrupts and page-flip events.
3. **Encoders (`struct drm_encoder`):** Abstract display serializers and protocol converters. They convert digital parallel RGB/YUV pixel data from CRTCs into protocol formats (TMDS for HDMI, packetized streams for DisplayPort, or LVDS/MIPI DSI).
4. **Connectors (`struct drm_connector`):** Represent physical output ports (HDMI-A, DP, eDP). Track physical connection status, drive Hot-Plug Detect (HPD) lines, query display identification metadata (EDID/DisplayID), and handle I2C/DDC communication buses.

Routing Multiplexers & Crossbar Bitmasks

Flexible crossbar switches link CRTC's, Encoders, and Connectors. The driver informs userspace of valid routing paths using 32-bit bitmasks:

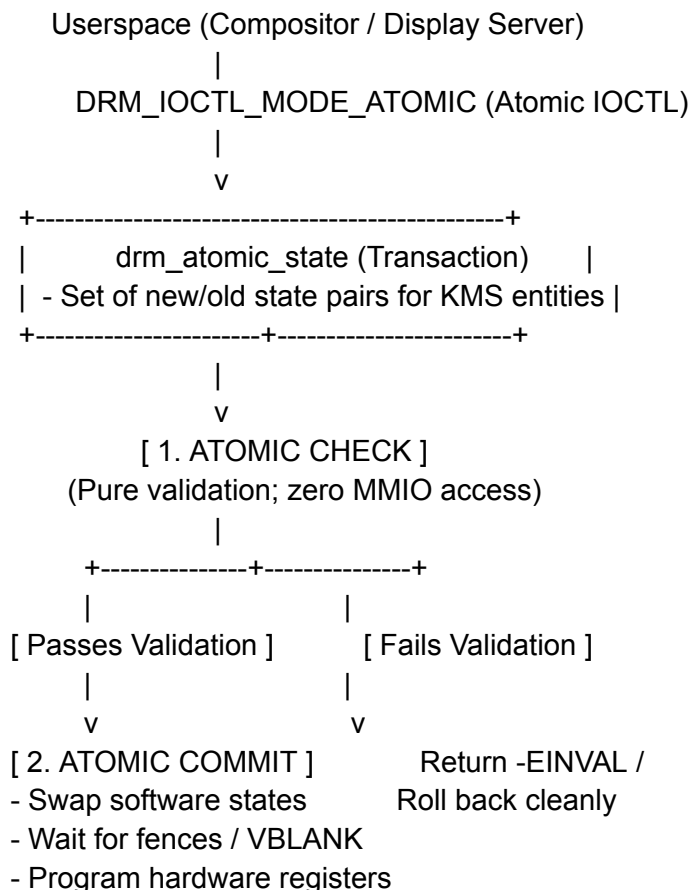
- `encoder->possible_crtcs`: Bitmask specifying which CRTC's can drive a given encoder. The N-th bit corresponds to the CRTC index in `dev->mode_config.crtc_list`: $\text{possible_crtcs} = \sum_i \text{in } \text{Supported CRTC's} \cdot 2^i$
- `plane->possible_crtcs`: Bitmask defining which CRTC composition engines a plane can feed.

7. The Atomic Modesetting Engine

Imperative vs. Transactional Atomic Modesetting

Legacy KMS executed modesets via piecewise, imperative IOCTL commands (`DRM_IOCTL_MODE_SETCRTC`, `SETPLANE`, `CURSOR`). This fragmented approach caused visual tearing, black frames, and broken intermediate states if an overlay assignment failed midway through an update.

The **Atomic Modesetting Engine** replaces legacy commands with transactional, all-or-nothing updates executed through `DRM_IOCTL_MODE_ATOMIC`.



Topography of Atomic State Objects

In the atomic framework, KMS entities (`drm_plane`, `drm_crtc`, `drm_connector`) are stateless wrappers. All operating parameters reside strictly within isolated state objects:

```

+-----+
|          struct drm_atomic_state          |
| +-----+ |
| | crtcs[]    -> [ struct drm_crtc_state    ] (Old / New)    | |
| | planes[]   -> [ struct drm_plane_state   ] (Old / New)    | |
| | connectors[] -> [ struct drm_connector_state ] (Old / New)    | |
| | private_objs -> [ struct drm_private_obj  ] (Old / New)    | |
| |                                     | |
| |                                     | |
| | Flags: DRM_MODE_ATOMIC_NONBLOCK, DRM_MODE_ATOMIC_TEST_ONLY | |
| | Global lock acquisition context: struct drm_modeset_acquire_ctx | |
| +-----+ |
+-----+

```

Drivers track proprietary silicon state (such as custom PLL dividers or FIFO watermarks) by subclassing base state objects:

```

+-----+
| struct minidrm_crtc_state |
| +-----+ |
| | struct drm_crtc_state base; | |
| | | |
| | /* Driver-Private Silicon Parameters */ | |
| | u32 pll_feedback_divider; | |
| | u32 pll_post_divider; | |
| | u32 fifo_watermark_low; | |
| | u32 fifo_watermark_high; | |
| | u64 required_dram_bandwidth_bps; | |
| +-----+ |
+-----+

```

The Two-Phase Transaction Lifecycle

An atomic update proceeds in two distinct steps:

1. **Phase 1: Validation (`atomic_check`)**
 - Executes pure software analysis across proposed states.
 - Verifies encoder/CRTC crossbar routing paths and calculates required PLL clock dividers.

- Aggregates total memory bandwidth across all active planes, rejecting updates that exceed bus limits with `-ENOSPC`.
- Clips plane crop coordinates against display viewports.
- **Crucial Rule:** Must remain completely read-only. `atomic_check` must **never** write to hardware MMIO or wait on hardware fences. If `DRM_MODE_ATOMIC_TEST_ONLY` is set, the transaction halts here and returns 0.

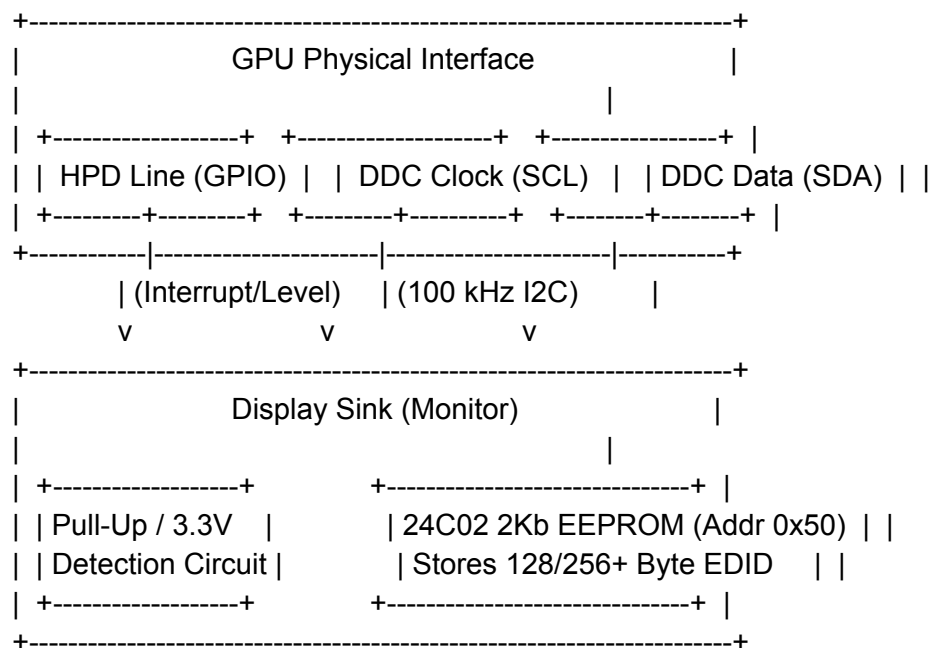
2. Phase 2: Execution (`atomic_commit`)

- Swaps software state pointers via `drm_atomic_helper_swap_state()`, making proposed states immediately active for software lookups.
- If `DRM_MODE_ATOMIC_NONBLOCK` is flagged, offloads hardware execution to a background workqueue and returns control to userspace immediately.
- Waits for upstream rendering fences (`dma_fence`) to signal before touching scanout registers.
- Programs MMIO registers during the vertical blanking window and arms VBLANK completion events.

8. Connector Subsystems, Hotplug Detection, & EDID Parsing

Hotplug Detection (HPD) Infrastructure

The connector subsystem handles dynamic external display connections, disconnections, and power cycles. In HDMI and DVI displays, connection state is indicated by an electrical High-Level signal (+5V) stepped down to 3.3V logic) on the HPD pin (Pin 19 in HDMI Type-A).



Line Debouncing & DP IRQ_HPD

Mechanical cable connection generates electrical contact bounce lasting 10\text{ ms} to 150\text{ ms}. To avoid interrupt storms, top-half interrupt handlers schedule a delayed workqueue enforcing a 150\text{ ms} to 250\text{ ms} software debounce delay before reading line status or probing the I2C bus.

In DisplayPort connections, sustained high signals (>100\text{ ms}) indicate monitor connection, while short low pulses (0.5\text{ ms} to 1.0\text{ ms}) represent an **IRQ_HPD**. IRQ_HPD signals an in-band interrupt requesting immediate service over the DisplayPort AUX channel (e.g., clock recovery loss or Multi-Stream Transport changes).

Display Data Channel (DDC) & EDID Decoding

Capabilities are queried across the Display Data Channel (DDC), a standard 100\text{ kHz} I²C bus where the GPU acts as Master and the display sink acts as Slave at address 0x50.

The sink exposes a 128-byte base VESA Extended Display Identification Data (EDID) block:

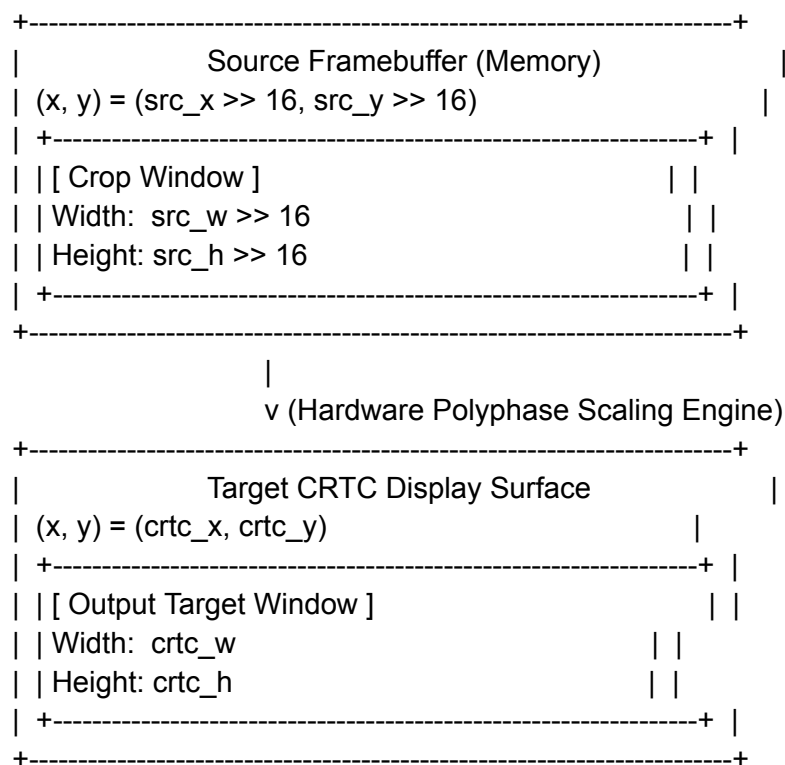
Byte Offset	Field Description	Validation Constraint / Parsing Rule
0x00 - 0x07	Fixed Header Pattern	Must match magic byte sequence 00 FF FF FF FF FF FF 00.
0x08 - 0x0B	Manufacturer & Product ID	3-character compressed ASCII vendor code and product ID.
0x12 - 0x13	EDID Version & Revision	Typically Version 1 Revision 3 or 4.
0x36 - 0x7F	Descriptor Blocks	Up to 4 detailed timing descriptors (DTDs) specifying native timings.
0x7E	Extension Block Flag	Byte count of additional 128-byte extension blocks (e.g., CEA-861 for HDR/Audio, DisplayID for high refresh rates).

0x7F	Block Checksum	Modulo-256 sum of all 128 bytes must evaluate to zero: $\sum_{i=0}^{127} \text{Byte}[i] \equiv 0 \pmod{256}$.
------	----------------	---

9. Plane Composition, Pixel Formats, & Color Management

Polyphase Scaling & Geometry Transformation

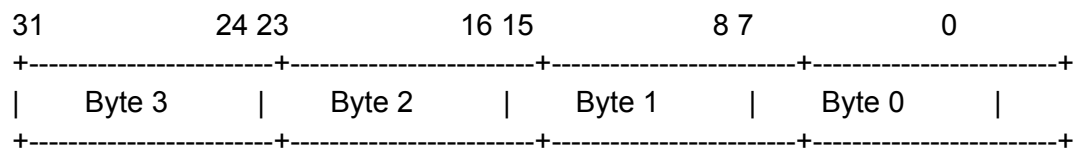
Plane hardware composites independent surfaces in memory directly onto active CRTC viewports:



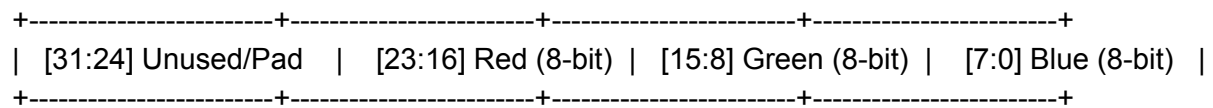
Source coordinates (*src_x*, *src_y*, *src_w*, *src_h*) are expressed in **16.16 fixed-point format** to allow sub-pixel crop panning. Destination coordinates (*crtc_x*, *crtc_y*, *crtc_w*, *crtc_h*) represent absolute integer pixel coordinates within the output display raster.

Pixel Formats (FourCC) & Memory Layouts

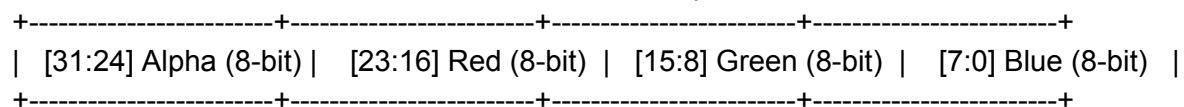
Pixel formats are identified globally by 32-bit DRM FourCC codes ([<uapi/drm/drm_fourcc.h>](#)):



DRM_FORMAT_XRGB8888 (Little-Endian in Memory):



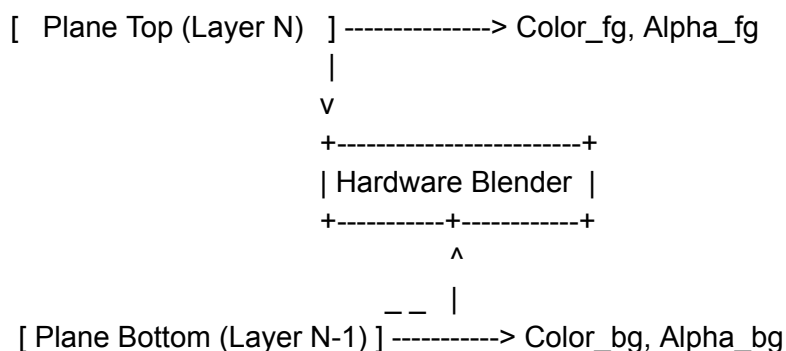
DRM_FORMAT_ARGB8888 (Little-Endian in Memory):



- **Packed Layouts (DRM_FORMAT_XRGB8888):** Color channels reside packed within a single contiguous memory buffer (1 plane).
- **Semi-Planar YUV 4:2:0 Layouts (DRM_FORMAT_NV12):** Memory is split across two planes. Plane 0 contains Luma (Y) samples at full resolution. Plane 1 contains interleaved Chroma (UV) samples subsampled 2×2. Cropping offsets must align to even-integer boundaries ($x \bmod 2 = 0$, $y \bmod 2 = 0$).

Hardware Composition & Alpha Blending

When planes overlap, hardware blenders calculate per-pixel composition based on standardized properties:



- **Layer Stacking (zpos / normalized_zpos):** Defines plane layer order from 0 (bottom-most background) to K-1 (top-most foreground).
- **Plane Global Alpha (alpha):** A 16-bit scalar ($0 \times 0000 = \text{transparent}$, $0 \times \text{FFFF} = \text{opaque}$) that scales per-pixel alpha values.
- **Premultiplied Blend Mode (DRM_MODE_BLEND_PREMULTI):** Evaluates:
 $C_{\text{out}} = C_{\text{fg}} + C_{\text{bg}} \times (1 - \alpha_{\text{fg}})$
- **Coverage Blend Mode (DRM_MODE_BLEND_COVERAGE):** Evaluates:
 $C_{\text{out}} = C_{\text{fg}} \times \alpha_{\text{fg}} + C_{\text{bg}} \times (1 - \alpha_{\text{fg}})$

Color Management Pipeline

The display engine processes pixels through a three-stage hardware color pipeline.

1. **Degamma LUT (DEGAMMA_LUT)**: Converts non-linear sRGB or Perceptual Quantizer (PQ) values into linear photometric space.
2. **Color Transformation Matrix (CTM)**: Applies a 3×3 affine matrix transformation in linear space for gamut remapping or YUV-to-RGB conversion. Represented in `struct drm_color_ctm` as S31.32 sign-magnitude fixed-point integers (Bit 63 = Sign, Bits [62:32] = Integer, Bits [31:0] = Fraction):
$$\begin{bmatrix} R_{\text{out}} \\ G_{\text{out}} \\ B_{\text{out}} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} R_{\text{in}} \\ G_{\text{in}} \\ B_{\text{in}} \end{bmatrix}$$
3. **Gamma LUT (GAMMA_LUT)**: Applies display transfer functions prior to signal output.

10. VBLANK Infrastructure, Precision Timestamps, & Page Flipping

VBLANK Core Architecture & Refcounted Interrupts

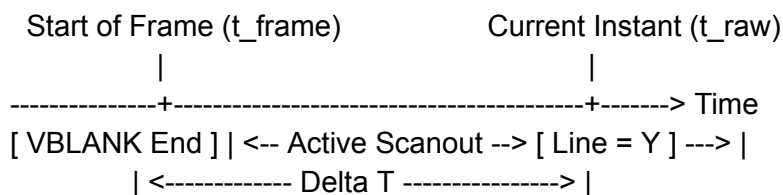
VBLANK interrupts execute at 60 Hz to 240 Hz+ per display pipe. servicing unneeded interrupts wastes power and prevents CPU cores from entering sleep states.

The DRM core enforces a **reference-counted interrupt gating** mechanism:

- When userspace requests VBLANK notifications, DRM invokes `drm_vblank_get()`, calling `.enable_vblank()` to unmask hardware MMIO interrupt bits when the counter transitions 0 to 1.
- When events complete, `drm_vblank_put()` decrements the count. A timer disables the physical interrupt if the counter remains at zero.

High-Precision Timestamps via Scanline Back-Projection

Sampling `ktime_get()` inside the ISR introduces microsecond-to-millisecond jitter due to interrupt latency. To achieve nanosecond-accurate presentation timestamps, DRM back-projects time by reading real-time hardware raster scanline counters:



Calculation: $t_{\text{frame}} = t_{\text{raw}} - \Delta T$

where: $\Delta T = (\text{Line_Current} * H_Total + \text{Pixel_Current}) / \text{Pixel_Clock}$

When `drm_crtc_handle_vblank()` fires, the driver reads instantaneous vertical line Y_{pos} and horizontal pixel X_{pos} from hardware registers:

- If scanout is within active video ($0 \leq Y < V_{\text{display}}$): $\Delta t = \frac{Y_{\text{pos}} \times H_{\text{total}} + X_{\text{pos}}}{\text{PixelClock}_{\text{Hz}}} t_{\text{vblank}} = t_{\text{sample}} - \Delta t$
- If scanout is within blanking ($V_{\text{display}} \leq Y < V_{\text{total}}$), Y_{pos} is converted to a negative offset relative to the next frame start ($Y_{\text{pos}} - V_{\text{total}}$), projecting forward in time.

High-Precision Page Flipping Mechanics

To present rendered frames without tearing:

1. Userspace: atomic commit with `DRM_MODE_PAGE_FLIP_EVENT`
|
2. Driver `atomic_flush / update`:
 - Program shadow registers with new FB base address
 - Arm hardware double-buffer flip trigger
 |
3. Driver `atomic_commit_tail`:
 - `drm_crtc_arm_vblank_event(crtc, event)`
 |
- +--- GPU Scanout Pipeline Continues ... ---+
4. Hardware hits VBLANK:
 - Latches shadow registers -> New FB scanned out
 - Hardware generates VBLANK IRQ
 |
5. Top-Half ISR:
 - `drm_crtc_handle_vblank(crtc)`
 - * Back-projects precision timestamp
 - * Matches armed pending events
 - * Dispatches `DRM_EVENT_FLIP_COMPLETE` to userspace with precision timestamp

11. Memory Backing, Framebuffers, & Dumb Buffer Allocations

Scanout DMA Engine Dynamics & Watermarks

Display DMA engines fetch pixels into hardware FIFOs to absorb memory bus latency. Transfers are regulated by watermarks:

Scanline byte stride (pitch) must align to hardware DMA burst alignment constraints (e.g., 64, 128, or 256 bytes):

$$\text{Pitch (Bytes)} = \text{ALIGN}(\text{Width} \times \text{BytesPerPixel}, \text{BURST_ALIGNMENT})$$

Framebuffer Objects (`struct drm_framebuffer`)

`struct drm_framebuffer` bridges memory allocations to display pipelines. It tracks geometry, format information (`struct drm_format_info`), memory strides (`pitches[]`), plane byte offsets (`offsets[]`), format modifiers, and holds refcounted pointers to backing GEM memory objects (`obj[]`).

The Dumb Buffer API Architecture

The Dumb Buffer API provides a hardware-agnostic method to allocate display buffers for unaccelerated software rendering, boot loaders, and fallback display compositors.

Backing Strategy Comparison

- **DMA / CMA Backed (`dma_alloc_wc()`):** Used by SoC display controllers lacking an IOMMU. Allocates physically contiguous, write-combined memory. Requires no manual CPU cache flushing.
- **SHMEM Backed (`shmem_file_setup()`):** Used by PCIe GPUs featuring IOMMUs or GARTs. Allocates pageable, cached system memory. Requires CPU cache maintenance (`clflush` or `dma_sync_single_for_cpu`).

12. Architectural Verification & Diagnostics Matrix

To verify driver state, inspect registered entity graph nodes, and debug operational issues, administrators and graphics developers can query the kernel diagnostic tools outlined below:

Inspect active DRM devices, minors, and assigned bus IDs

```
$ cat /sys/kernel/debug/dri/0/name
custom_drm dev=0000:01:00.0 unique=0000:01:00.0
```

Inspect active client connections and authentication status

```
$ cat /sys/kernel/debug/dri/0/clients
command pid dev master a uid magic
gnome-shell 1842 0 y y 1000 0
wayland 2014 128 n y 1000 0
```

Check active PCIe device link parameters and MSI-X vector allocation

```
$ lspci -v -s 0000:01:00.0
01:00.0 VGA compatible controller: Custom Silicon Accelerator
Flags: bus master, fast devsel, latency 0, IRQ 125
Memory at f6000000 (32-bit, non-prefetchable) [size=16M]
```

Memory at e0000000 (64-bit, prefetchable) [size=256M]
Capabilities: [MSI-X] Vector table: BAR=0, offset=0x00002000

Inspect active hardware interrupt execution rates

\$ cat /proc/interrupts | grep custom_gpu

125: 1042 0 PCI-MSI-X 524288-edge custom_gpu_engine_irq

Interrogate global KMS atomic pipeline state

\$ cat /sys/kernel/debug/dri/0/state

Inspect display hardware entity graph parameters via modetest

\$ modetest -M custom_gpu -p

Planes:

id	crtc	fb	CRTC x,y	x,y	gamma	size	formats
----	------	----	----------	-----	-------	------	---------

31	33	0	0,0	0,0	0		XR24, AR24, NV12
----	----	---	-----	-----	---	--	------------------

CRTCs:

id	pipe	mode
----	------	------

33	0	1920x1080@60
----	---	--------------

Encoders:

id	crtc	type	possible_crtcs	possible_clones
----	------	------	----------------	-----------------

34	33	TMDs	0x00000001	0x00000000
----	----	------	------------	------------

Connectors:

id	encoder	status	name	modes
----	---------	--------	------	-------

35	34	connected	HDMI-A-1	1920x1080@60
----	----	-----------	----------	--------------