LINUX DRM

Introduction

- **Direct Rendering Manager** (**DRM**) is a subsystem of the Linux kernel that provides an interface with the GPUs.
- DRM exposes an API that user-space programs can use to send commands and data to the GPU and perform configuration operations such as the mode settings of the display.

History

- DRM was first developed as the kernel space component of the Xserver's Rendering Infrastructure
- Since then it has been used by other graphic stack alternatives such as Wayland.
- User-space programs can use the DRM API to command the GPU to do hardware accelerated 3D rendering and as well as GPGPU computing.

History

- fbdev
  - The Linux kernel API that allowed to manage the framebuffer of a graphics adapter
  - Could not handle the needs of modern 3D accelerated GPU based infrastructure.
- GPU infrastructure based hardware requires
  - setting and managing a command queue in the card's memory (Video RAM) to dispatch commands to the GPU.
  - a proper management of the buffers and free space of the Video RAM

History

- User space programs (such as the X server) directly managed these resources.
  - Exercised access to the card's resources as though exclusive.
  - When two or more programs tried to control the same video card at the same time, and set its resources each one in its own way, most times they ended catastrophically

Evolution of DRM

- When the Direct Rendering Manager was first created, the purpose was that multiple programs using resources from the video card can cooperate through it.
- The DRM gets an exclusive access to the video card, and it's responsible for initializing and maintaining the command queue, the VRAM and any other hardware resource.

Evolution of DRM

- The programs that want to use the GPU send their requests to DRM, which acts as an arbitrator and takes care to avoid possible conflicts.
- Scope of DRM has been expanded over the years to cover more functionality previously handled by user space programs, such as framebuffer managing and mode-setting, memory sharing objects and memory synchronization.

Evolution of DRM

- Some of these expansions had carried their own specific names, such as Graphics Execution Manager (GEM) or Kernel Mode-Setting (KMS), and the terminology prevails when the functionality they provide is specifically alluded.
  - All are really parts of the whole kernel DRM subsystem.

Evolution of DRM

- The trend to include two GPUs in a computer —a discrete GPU and an integrated one— led to new problems such as GPU Switching to be also solved at the DRM layer.
  - DRM was provided with GPU offloading abilities, called PRIME

Software Architecture

- The Direct Rendering Manager resides in kernel space, so the user space programs must use kernel system calls to request its services.
- DRM doesn't define its own customized system calls.
- Exposes the GPUs through the filesystem name space using device files under the /dev hierarchy.

Software Architecture

- Each GPU detected by DRM is referred as a *DRM device*, and a device file /dev/dri/card*X* (where *X* is a sequential number) is created to interface with it.
- User space programs that want to talk to the GPU must open the file and use ioctl calls to communicate with DRM.
- Different ioctls correspond to different functions of the DRM API.

Software Architecture

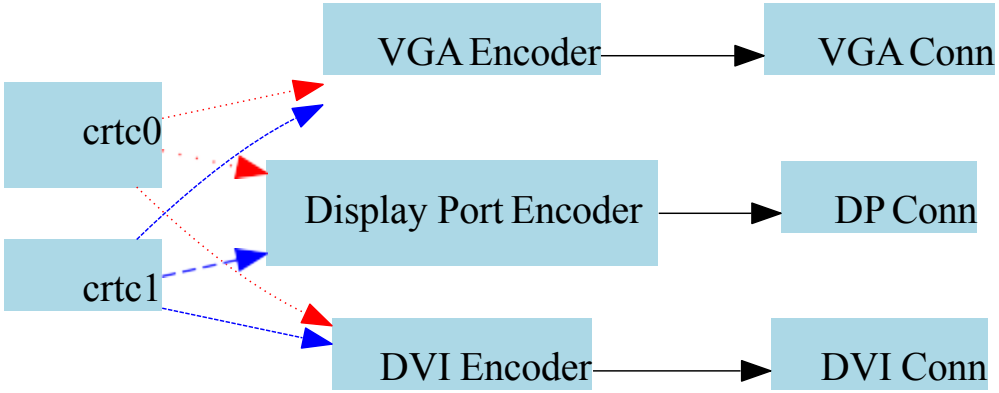
- A library called *libdrm* was created to facilitate the interface of user space programs with the DRM subsystem.
- This library is merely a wrapper that provides a function written in C for every ioctl of the DRM API, as well as constants, structures and other helper elements.

- DRM consists of two parts: a generic "DRM core" and a specific one ("DRM Driver") for each type of supported hardware.
- DRM core provides the basic framework where different DRM drivers can register, and also provides to user space a minimum set of ioctls with common, hardware-independent functionality.

Software Architecture

- A DRM driver implements the hardware-dependent part of the API, specific to the type of GPU it supports;
  - Provides the implementation of the remaining ioctls not covered by DRM core.
  - Extends the API offering additional ioctls with extra functionality only available on such hardware.
  - When a specific DRM driver provides an enhanced API, user space libdrm is also extended by an extra library libdrm-*driver* that can be used by user space to interface with the additional ioctls.

DRM Components

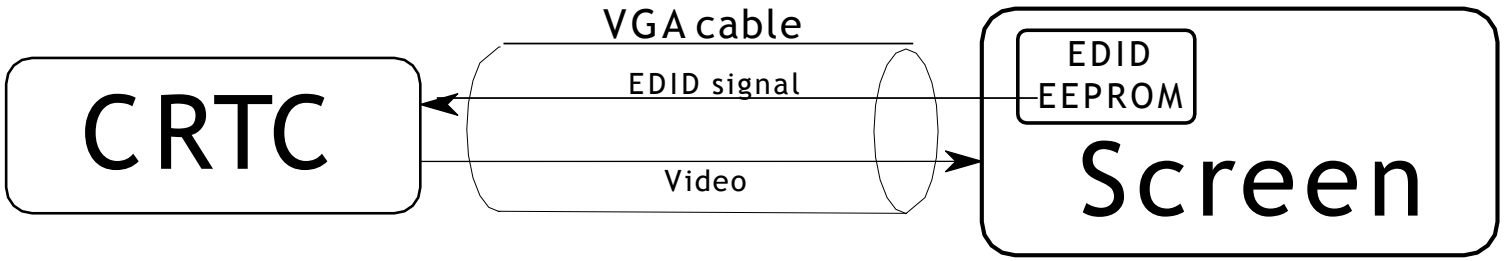


Framebuffer: The image to be displayed on the screen(VRAM)

CRTC: Streams the framebuffer following the screen's timings

Encoder: Convert the CRTC's output to the right PHY signal

Connector: The actual connector where the screen is plugged



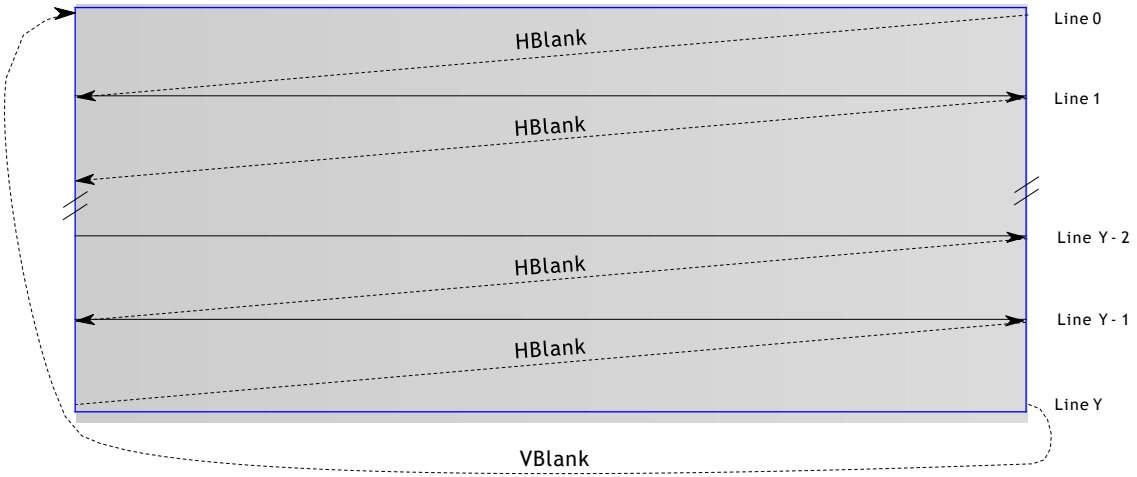
Stored in each connector of the screen (small EEPROM)

Is usually accessed via a dedicated I2C line in the connector

Holds the modes supported by the screen connector

Processed by the host driver and exposed with the tool

CRTC SCANOUT



- Streams the framebuffer following the screen's timings
- After each line, the CRTC must wait for the CRT to go back to the beginning of the next line (Horizontal Blank)
- After each frame, the CRTC must wait for the CRT to go back to the first line (Vertical Blank)
- Timings are met by programming the CRTC clock using PLLs

DRM and libdrm

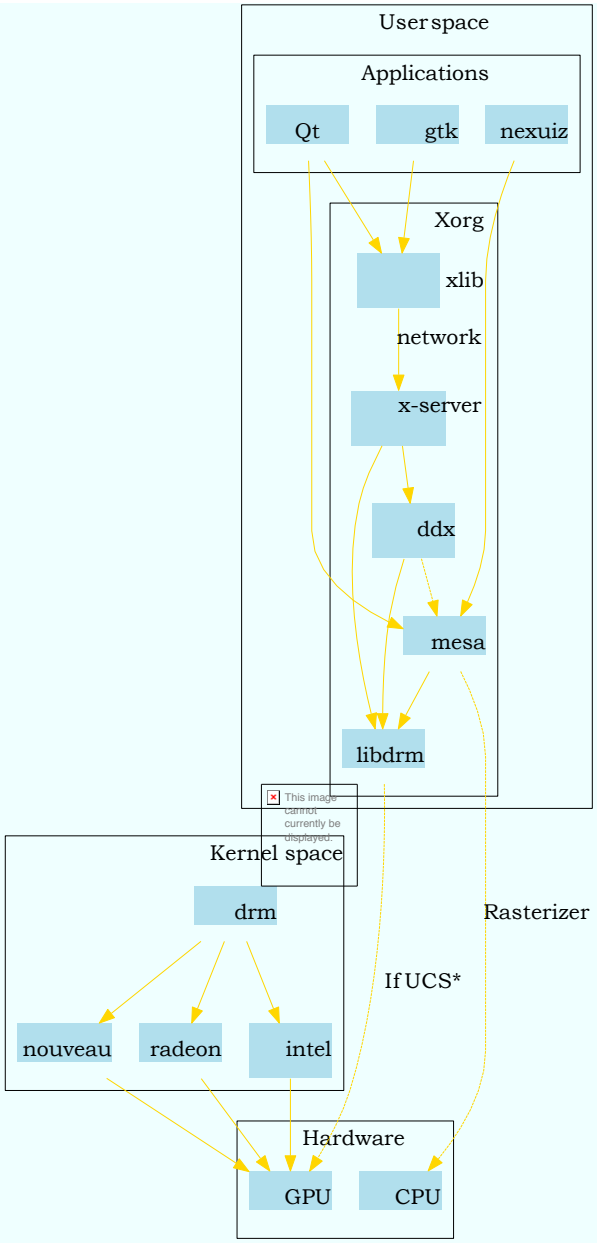
DRM

- Inits and configures the GPU;
- Performs Kernel Mode Setting (KMS);
- Exports privileged GPU primitives:
  - Create context + VM allocation;
  - Command submission;
  - VRAM memory management: GEM & TTM;
  - Buffer-sharing: GEM & DMA-Buf;
- Implementation is driver-dependent.

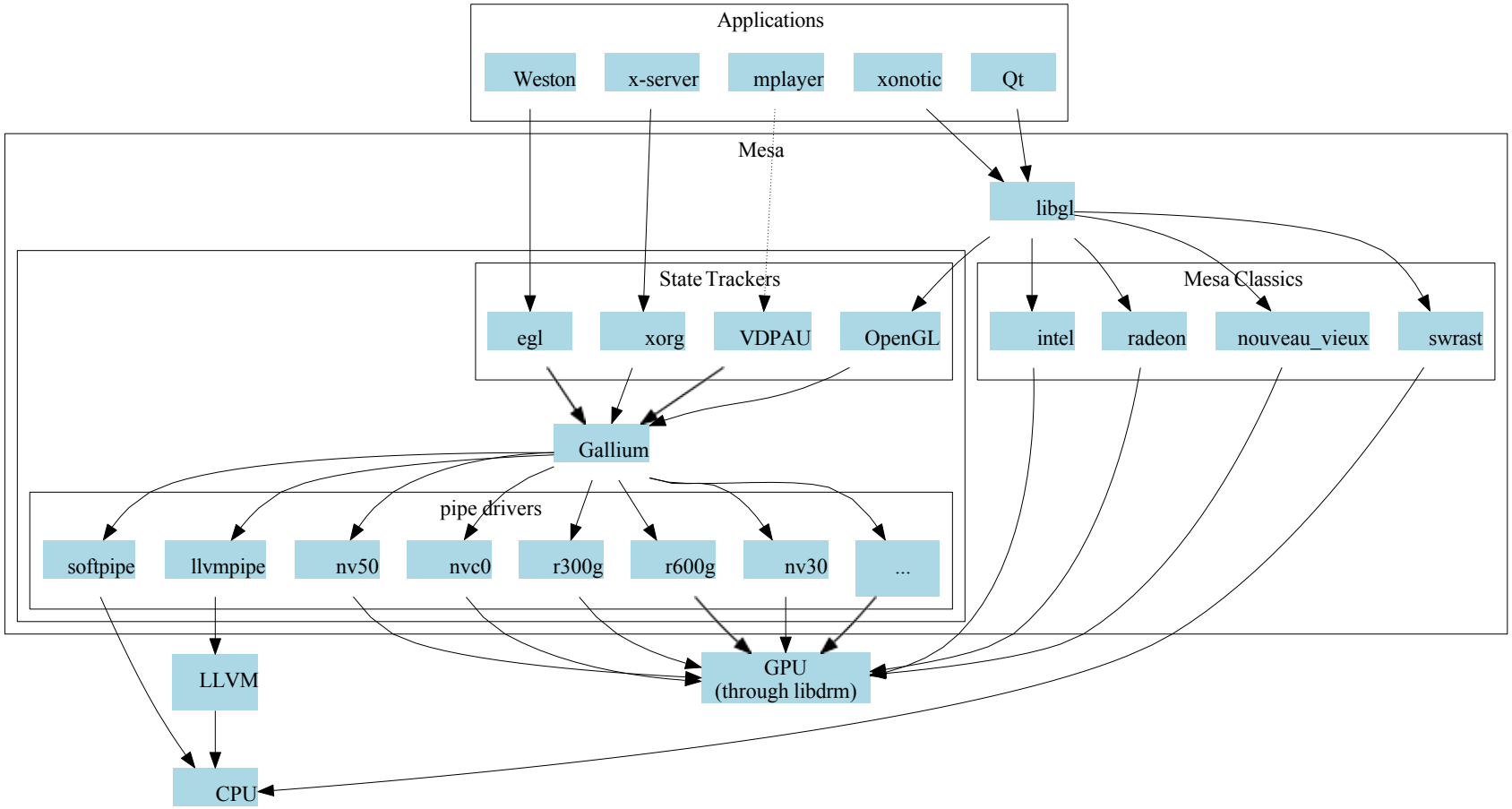
libdrm

- Wraps the DRM interface into a usable API;
  - Is meant to be only used by Mesa & the DDX;

DRM and libdrm



DRM in the Larger Canvas



Introduction

- The Linux DRM layer contains code intended to support the needs of complex graphics devices, usually containing programmable pipelines well suited to 3D graphics acceleration.
- Graphics drivers in the kernel may make use of DRM functions to make tasks like memory management, interrupt handling and DMA easier, and provide a uniform interface to applications.

- At the core of every DRM driver is a struct drm_driver structure.
- Drivers typically statically initialize a drm_driver structure, and then pass it to drm_dev_alloc( ) to allocate a device instance.
- After the device instance is fully initialized it can be registered (which makes it accessible from userspace) using drm_dev_register().

- The struct drm_driver structure contains static information that describes the driver and features it supports, and pointers to methods that the DRM core will call to implement the DRM API.

- **DRIVER_USE_AGP**
  - Driver uses AGP interface, the DRM core will manage AGP resources.
- **DRIVER_LEGACY**
  - Denote a legacy driver using shadow attach.
  - Deprecated
- **DRIVER_KMS_LEGACY_CONTEXT**
  - Used only by nouveau for backwards compatibility with existing userspace.
  - Deprecated.

- **DRIVER_PCI_DMA**
  - Driver is capable of PCI DMA, mapping of PCI DMA buffers to userspace will be enabled. Deprecated.
- **DRIVER_SG**
  - Driver can perform scatter/gather DMA, allocation and mapping of scatter/gather buffers will be enabled. Deprecated.
- **DRIVER_HAVE_DMA**
  - Driver supports DMA, the userspace DMA API will be supported. Deprecated.

- **DRIVER_HAVE_IRQ;**
  DRIVER_HAVE_IRQ indicates whether the driver has an IRQ handler managed by the DRM Core.
  - The core will support simple IRQ handler installation when the flag is set.
- **DRIVER_IRQ_SHARED**
  - DRIVER_IRQ_SHARED indicates whether the device & handler support shared IRQs (note that this is required of PCI drivers).

- **DRIVER_GEM**
  - Driver use the GEM memory manager.
- **DRIVER_MODESET**
  - Driver supports mode setting interfaces (KMS).
- **DRIVER_PRIME**
  - Driver implements DRM PRIME buffer sharing.
- **DRIVER_RENDER**
  - Driver supports dedicated render nodes.

- **DRIVER_ATOMIC**
  - Driver supports atomic properties.
  - In this case the driver must implement appropriate obj->atomic_get_property() vfuncs for any modeset objects with driver specific properties.

Major, Minor and Patchlevel

- int major; int minor; int patchlevel;
- The DRM core identifies driver versions by a major, minor and patch level triplet.
- The information is printed to the kernel log at initialization time and passed to userspace through the DRM_IOCTL_VERSION ioctl.

- The major and minor numbers are also used to verify the requested driver API version passed to DRM_IOCTL_SET_VERSION.
- When the driver API changes between minor versions, applications can call DRM_IOCTL_SET_VERSION to select a specific version of the API.

- If the requested major isn't equal to the driver major, or the requested minor is larger than the driver minor, the DRM_IOCTL_SET_VERSION call returns an error else the driver's set_version() method will be called with the requested version.

Name, Description and Date

```
char *name;
char *desc;
char *date;
```

- The driver name is printed to the kernel log at initialization time, used for IRQ registration and passed to userspace through DRM_IOCTL_VERSION.
- The driver description is a purely informative string passed to userspace through the DRM_IOCTL_VERSION ioctl otherwise unused by the kernel.

Driver Load

- The load method is the driver and device initialization entry point.
- The method is responsible for
  - allocating and initializing driver private data
  - specifying supported performance counters
  - performing resource allocation and mapping
    - acquiring clocks,
    - mapping registers
    - allocating command buffers

- initializing the memory manager
- installing the IRQ handler
- setting up vertical blanking handling
- mode setting
- initial output configuration

Device Instance and Driver Handling

- A device instance for a drm driver is represented by struct drm_device.
- This is allocated with drm_dev_alloc(), usually from bus-specific ->:c:func:*probe()* callbacks implemented by the driver.
- The driver then needs to initialize all the various subsystems for the drm device like memory management, vblank handling, modesetting support and intial output configuration and initialize all the corresponding hardware bits

- Call to drm_dev_set_unique() sets the userspace-visible unique name of the device instance.
- Finally when everything is up and running and ready for userspace the device instance can be published using drm_dev_register()

Clean up

- Clean up operations
  - Unpublish the device instance with drm_dev_unregister().
  - Clean up any other resources allocated at device initialization and drop the driver's reference to drm_device using drm_dev_unref()

Driver Information

- **Driver Features**
  - Drivers inform the DRM core about their requirements and supported features by setting appropriate flags in the *driver_features* field.
  - flags influence the DRM core behaviour since registration time, most of them must be set to registering the drm_driver instance.

u32 driver_features

- **Driver Feature Flags**
- DRIVER_USE_AGP
  - Driver uses AGP interface, the DRM core will manage AGP resources.
- DRIVER_REQUIRE_AGP
  - Driver needs AGP interface to function. AGP initialization failure will become a fatal error.
- DRIVER_SGDriver can perform scatter/gather DMA, allocation and mapping of scatter/gather buffers will be enabled. Deprecated.

struct drm_driver

```
struct drm_driver {
  int (* load) (struct drm_device *, unsigned long flags);
  void (* unload) (struct drm_device *);
  void (* release) (struct drm_device *);
  u32 (* get_vblank_counter) (struct drm_device *dev, unsigned int pipe);
  int (* enable_vblank) (struct drm_device *dev, unsigned int pipe);
  void (* disable_vblank) (struct drm_device *dev, unsigned int pipe);
  int (* get_scanout_position) (struct drm_device *dev, unsigned int pipe,unsigned int flags
int *vpos, int *hpos,ktime_t *stime, ktime_t *etime,const struct drm_display_mode *mode);
  int (* get_vblank_timestamp) (struct drm_device *dev, unsigned int pipe,int
*max_error,struct timeval *vblank_time,unsigned flags);
  int (* master_create) (struct drm_device *dev, struct drm_master *master);
  void (* master_destroy) (struct drm_device *dev, struct drm_master *master);
  int (* master_set) (struct drm_device *dev, struct drm_file *file_priv,bool from_open);
  void (* master_drop) (struct drm_device *dev, struct drm_file *file_priv);
  void (* gem_free_object) (struct drm_gem_object *obj);
  void (* gem_free_object_unlocked) (struct drm_gem_object *obj);
  struct drm_gem_object *(* gem_create_object) (struct drm_device *dev,size_t size);
  int (* dumb_create) (struct drm_file *file_priv,struct drm_device *dev,struct
drm_mode_create_dumb *args);
  int (* dumb_map_offset) (struct drm_file *file_priv,struct drm_device *dev, uint32_t
handle,uint64_t *offset);
  int (* dumb_destroy) (struct drm_file *file_priv,struct drm_device *dev,uint32_t handle);
};
```

load

- Backward-compatible driver callback to complete initialization steps after the driver is registered.
- May suffer from race conditions and its use is deprecated for new drivers.
- Supported only for existing drivers not yet converted to the new scheme.
- drm_dev_init and drm_dev_register should be used for race-free set up of a struct drm_device.

unload

- Reverse the effects of the driver load callback.
- Ideally, the clean up performed by the driver should happen in the reverse order of the initialization.
- Similar to the load hook, this handler is deprecated and its usage should be dropped in favor of an open-coded teardown function at the driver layer.
- drm_dev_unregister() and drm_dev_unref() for the proper way to remove a struct drm_device.
- The unload() hook is called right after unregistering the device.

release

- Optional callback for destroying device data after the final reference is released, i.e. the device is being destroyed.
- Drivers using this callback are responsible for calling drm_dev_fini() to finalize the device and then freeing the struct themselves.

get_vblank_counter

- Driver callback for fetching a raw hardware vblank counter for the CRTC specified with the pipe argument.
- If a device doesn't have a hardware counter, the driver can simply use drm_vblank_no_hw_counter() function.
- The DRM core will account for missed vblank events while interrupts where disabled based on system timestamps.

- Wraparound handling and loss of events due to modesetting is dealt with in the DRM core code, as long as drivers call drm_crtc_vblank_off() and drm_crtc_vblank_on()when disabling or enabling a CRTC.

enable_vblank

- Enable vblank interrupts for the CRTC specified with the pipe argument.
- Returns:
  - Zero on success, appropriate errno if the given **crtc**‘s vblank interrupt cannot be enabled.

disable_vblank

- Disable vblank interrupts for the CRTC specified with the pipe argument.

get_scanout_position

- Called by vblank timestamping code.
- Returns the current display scanout position from a crtc, and an optional accurate ktime_get() timestamp of when position was measured.
- Helper callback which is only used if a driver uses
- drm_calc_vbltimestamp_from_scanoutpos() for the **get_vblank_timestamp** callback.

Parameters

- dev:
  - DRM device.
- pipe:
  - Id of the crtc to query.
- flags:
  - Flags from the caller (DRM_CALLED_FROM_VBLIRQ or 0).
- vpos:
  - Target location for current vertical scanout position.

- ≈

- hpos:
  - Target location for current horizontal scanout position.
- stime:
  - Target location for timestamp taken immediately before scanout position query.
  - Can be NULL to skip timestamp.

- etime:
  - Target location for timestamp taken immediately after scanout position query.
  - Can be NULL to skip timestamp
- mode:
  - Current display timings.

- Returns vpos as a positive number while in active scanout area.
- Returns vpos as a negative number inside vblank, counting the number of scanlines to go until end of vblank,
- e.g., -1 means “one scanline until start of active scanout / end of vblank.”

- Returns:
- Flags, or'ed together as follows:
- DRM_SCANOUTPOS_VALID:
  - Query successful.
- DRM_SCANOUTPOS_INVBL:
  - Inside vblank.
- DRM_SCANOUTPOS_ACCURATE
  - Returned position is accurate.
  - A lack of this flag means that returned position may be offset by a constant but unknown small number of scanlines wrt. real scanout position.

- get_vblank_timestamp
  - Called by drm_get_last_vbltimestamp().
  - Should return a precise timestamp when the most recent VBLANK interval ended or will end.
- The timestamp in **vblank_time** should correspond as closely as possible to the time when the first video scanline of the video frame after the end of VBLANK will start scanning out, the time immediately after end of the VBLANK interval.

- If the **crtc** is currently inside VBLANK, this will be a time in the future.
- If the **crtc** is currently scanning out a frame, this will be the past start time of the current scanout.
- This is meant to adhere to the OpenML OML_sync_control extension specification.

- Paramters:
- dev:
  - dev DRM device handle.
- pipe:
  - crtc for which timestamp should be returned.
- max_error:
  - Maximum allowable timestamp error in nanoseconds.
  - Implementation should strive to provide timestamp with an error of at most max_error nanoseconds.
  - Returns true upper bound on error for

  - timestamp.
- vblank_time:
  - Target location for returned vblank timestamp.
- flags:
  - 0 = Defaults, no special treatment needed.
  - DRM_CALLED_FROM_VBLIRQ = Function is called from vblank irq handler.
  - Some drivers need to apply some workarounds for gpu-specific vblank irq quirks if flag is set.

- Returns:
  - Zero if timestamping isn't supported in current display mode or a negative number on failure.
  - A positive status code on success, which describes how the vblank_time timestamp was computed.
- master_create
  - Called whenever a new master is created.
  - Only used by vmwgfx.

- master_destroy
  - Called whenever a master is destroyed.
  - Only used by vmwgfx.
- master_set
  - Called whenever the minor master is set.
  - Only used by vmwgfx.
- master_drop
  - Called whenever the minor master is dropped.
  - Only used by vmwgfx.

- gem_free_object
  - deconstructor for drm_gem_objects
  - deprecated and should not be used by new drivers.
  - **gem_free_object_unlocked** should be used instead.
- gem_free_object_unlocked
  - deconstructor for drm_gem_objects
  - This is for drivers which are not encumbered with drm_device.struct_mutex legacy locking schemes.
    - Use this hook instead of **gem_free_object**.

- gem_create_object
  - constructor for gem objects
  - Hook for allocating the GEM object struct, for use by core helpers.
- dumb_create
  - This creates a new dumb buffer in the driver's backing storage manager (GEM, TTM or something else entirely) and returns the resulting buffer handle.
  - This handle can then be wrapped up into a framebuffer modeset object.

- Userspace is not allowed to use such objects for render acceleration –
- drivers must create their own private ioctls for such a use case.
- Width, height and depth are specified in the drm_mode_create_dumb argument.
- The callback needs to fill the handle, pitch and size for the created buffer.
- Called by the user via ioctl.

- Returns:
  - Zero on success, negative errno on failure.
- dumb_map_offset
  - Allocate an offset in the drm device node's address space to be able to memory map a dumb buffer.
  - GEM-based drivers must use drm_gem_create_mmap_offset() to implement this.
- Called by the user via ioctl.
- Returns:
  - Zero on success, negative errno on failure.

- dumb_destroy
  - This destroys the userspace handle for the given dumb backing storage buffer. Since buffer objects must be reference counted in the kernel a buffer object won't be immediately freed if a framebuffer modeset object still uses it.
- Called by the user via ioctl.
- Returns:
  - Zero on success, negative errno on failure.

Description

- This structure represent the common code for a family of cards.
- one drm_device for each card present in this family.

Driver Private & Performance Counters

- The driver private hangs off the main drm_device structure and can be used for tracking various device-specific bits of information, like register offsets, command buffer status, register state for suspend/resume, etc.
- At load time, a driver may simply allocate one and set drm_device.*dev_priv* appropriately;
- it should be freed and drm_device.*dev_priv* set to NULL when the driver is unloaded.
-

- DRM supports several counters which were used for rough performance characterization.
- stat counter system is deprecated and should not be used.
- If performance monitoring is desired, the developer should investigate and potentially enhance the kernel perf and tracing infrastructure to export GPU related performance information for consumption by performance monitoring tools and applications.

Interrupts

IRQ Registration

- The DRM core tries to facilitate IRQ handler registration and unregistration by providing drm_irq_installand drm_irq_unin stall functions.
- Those functions only support a single interrupt per device, devices that use more than one IRQs need to be handled manually.

Managed IRQ Registration

- drm_irq_install and drm_irq_uninstall functions get the device IRQ by calling drm_dev_to_irq
- drm_dev_to_irq inline function calls a bus-specific operation to retrieve the IRQ number.
  - For platform devices, platform_get_irq(..., 0) is used to retrieve the IRQ number.
- drm_irq_install starts by calling the irq_preinstall driver operation.

- The operation is optional and must make sure that the interrupt will not get fired by clearing all pending interrupt flags or disabling the interrupt.
- IRQ is requested by a call to request_irq.
- If the DRIVER_IRQ_SHARED driver feature flag is set, a shared (IRQF_SHARED) IRQ handler will be requested.

- The IRQ handler function must be provided as the mandatory irq_handler driver operation.
- It will get passed directly to request_irq and has the same prototype as all IRQ handlers.
- It will get called with a pointer to the DRM device as the second argument.

- Finally the optional irq_postinstall driver operation function is called.
- The operation usually enables interrupts (excluding the vblank interrupt, which is enabled separately), but drivers may choose to enable/disable interrupts at a different time.
- drm_irq_uninstall is used to uninstall an IRQ handler.

- It starts by waking up all processes waiting on a vblank interrupt to make sure they don't hang, and then calls the optional irq_uninstall driver operation.
- The operation must disable all hardware interrupts. Finally the function frees the IRQ by calling free_irq.

Manual IRQ Registration

- Drivers that require multiple interrupt handlers cannot use the managed IRQ registration functions.
  - In that case IRQs must be registered and unregistered manually (usually with the request_irq and free_irq functions, or their devm_* equivalent).
  - When manually registering IRQs, drivers must not set the DRIVER_HAVE_IRQ driver feature flag, and must not provide the irq_handler driver operation.
  - They must set the drm_device *irq_enabled* field to 1 upon registration of the IRQs, and clear it to 0 after unregistering the IRQs.

Memory Management

- The DRM core includes two memory managers, namely Translation Table Maps (TTM) and Graphics Execution Manager (GEM).
- TTM was the first DRM memory manager to be developed and tried to be a one-size-fits-them all solution.
  - It provides a single userspace API to accommodate the need of all hardware, supporting both Unified Memory Architecture (UMA) devices and devices with dedicated video RAM (i.e. most discrete video cards).
  - This resulted in a large, complex piece of code hard to use for driver development.

GEM

- GEM started as an Intel-sponsored project in reaction to TTM's complexity
- instead of providing a solution to every graphics memory-related problems, GEM identified common code between drivers and created a support library to share it.
- GEM has simpler initialization and execution requirements than TTM, but has no video RAM management capabitilies and is limited to UMA devices.

- GEM exposes
  - a set of standard memory-related operations to userspace
  - a set of helper functions to drivers
  - drivers implement hardware-specific operations with their own private API.

- Data-agnostic.
- Content unaware management of abstract buffer objects.
- Buffer content based APIs such as buffer allocation or synchronization primitives, are out of the scope of GEM
  - must be implemented using driver-specific ioctls.

- GEM operations:
  - Memory allocation and freeing
  - Command execution
  - Aperture management at command execution time
- Buffer object allocation provided by Linux's shmem layer, which provides memory to back each object.
- Device-specific operations, such as command execution, pinning, buffer read & write, mapping, and domain ownership transfers are left to driver-specific ioctls.

GEM Initialization

- Drivers that use GEM must set the DRIVER_GEM bit in the struct drm_driver *driver_features* field.
- The DRM core automatically initializes the GEM core before calling the load operation.
  - this will create a DRM Memory Manager object which provides an address space pool for object allocation.
-

- In a KMS configuration, drivers need to allocate and initialize a command ring buffer following core GEM initialization if required by the hardware.
- UMA device’s "stolen" memory region provides space for the initial framebuffer and large, contiguous memory regions required by the device.
- This space is not managed by GEM, and must be initialized separately into its own DRM MM object.

GEM Objects Creation

- GEM splits creation of GEM objects and allocation of the memory that backs them in two distinct operations.
- GEM objects are represented by an instance of struct drm_gem_object.
- Drivers need to extend GEM objects with private information and thus create a driver-specific GEM object structure type that embeds an instance of struct drm_gem_object.

- To create a GEM object, a driver allocates memory for an instance of its specific GEM object type and initializes the embedded struct drm_gem_object with a call to drm_gem_object_init.
- The function takes a pointer to the DRM device, a pointer to the GEM object and the buffer object size in bytes.
- GEM uses shmem to allocate anonymous pageable memory.

- drm_gem_object_init will create an shmfs file of the requested size and store it into the struct drm_gem_object *filp* field.
- The memory is used as either main storage for the object when the graphics hardware uses system memory directly or as a backing store otherwise.
- Drivers are responsible for the actual physical pages allocation by calling shmem_read_mapping_page_gfp for each page.

- Drivers can decide to allocate pages when initializing the GEM object, or delay allocation until the memory is needed (for instance when a page fault occurs as a result of a userspace memory access or when the driver needs to start a DMA transfer involving the memory).

- Anonymous pageable memory allocation not always desired.
  - Particularly in embedded systems when the hardware requires physically contiguous system memory
- Drivers can create GEM objects with no shmfs backing (called private GEM objects) by initializing them with a call to drm_gem_private_object_init instead of drm_gem_object_init.
- Storage for private GEM objects must be managed by drivers.

- Drivers that do not need to extend GEM objects with private information can call the drm_gem_object_alloc function to allocate and initialize a struct drm_gem_object instance.
- The GEM core will call the optional driver gem_init_object operation after initializing the GEM object with drm_gem_object_init.
- int (*gem_init_object) (struct drm_gem_object *obj);
- No alloc-and-init function exists for private GEM objects.

GEM Objects Lifetime

- All GEM objects are reference-counted by the GEM core.
- References can be acquired and released by calling drm_gem_object_reference and drm_gem_object_unreference respectively.
- The caller must hold the drm_device *struct_mutex* lock.
- GEM provides the drm_gem_object_reference_unlocked and drm_gem_object_unreference_unlocked functions that can be called without holding the lock.

- When the last reference to a GEM object is released the GEM core calls the drm_driver gem_free_object operation.
- Freeing operation mandatory for GEM-enabled drivers which frees the GEM object and all associated resources.
- void (*gem_free_object) (struct drm_gem_object *obj);

- Drivers are responsible for freeing all GEM object resources, including the resources created by the GEM core.
- If an mmap offset has been created for the object (in which case drm_gem_object::*map_list*::*map* is not NULL) it must be freed by a call to drm_gem_free_mmap_offset.
- The shmfs backing store must be released by calling drm_gem_object_release (can safely be called if no shmfs backing store has been created).

GEM Objects Naming

- Communication between userspace and the kernel refers to GEM objects using local handles, global names or, file descriptors.
  - All of those are 32-bit integer values; the usual Linux kernel limits apply to the file descriptors.
- GEM handles are local to a DRM file.
- Applications get a handle to a GEM object through a driver-specific ioctl, and can use that handle to refer to the GEM object in other standard or driver-specific ioctls.

- Closing a DRM file handle frees all its GEM handles and dereferences the associated GEM objects.
- To create a handle for a GEM object drivers call drm_gem_handle_create.
- The function takes a pointer to the DRM file and the GEM object and returns a locally unique handle.
- When the handle is no longer needed drivers delete it with a call to drm_gem_handle_delete.
- Finally the GEM object associated with a handle can be retrieved by a call to drm_gem_object_lookup.

- Handles don't take ownership of GEM objects, they only take a reference to the object that will be dropped when the handle is destroyed.
- To avoid leaking GEM objects, drivers must make sure they drop the reference(s) they own (such as the initial reference taken at object creation time) as appropriate, without any special consideration for the handle.
  - For example, in the particular case of combined GEM object and handle creation in the implementation of the dumb_create operation, drivers must drop the initial reference to the GEM object before returning the handle.

- GEM names are similar in purpose to handles but are not local to DRM files.
  - They can be passed between processes to reference a GEM object globally.
  - Names can't be used directly to refer to objects in the DRM API, applications must convert handles to names and names to handles using the DRM_IOCTL_GEM_FLINK and DRM_IOCTL_GEM_OPEN ioctls respectively.
- The conversion is handled by the DRM core without any driver-specific support.

- Similar to global names, GEM file descriptors are also used to share GEM objects across processes.
  - They offer additional security: as file descriptors must be explicitly sent over UNIX domain sockets to be shared between applications, they can't be guessed like the globally unique GEM names.

- Drivers that support GEM file descriptors, also known as the DRM PRIME API, must set the DRIVER_PRIME bit in the struct drm_driver *driver_features* field, and implement the prime_handle_to_fd and prime_fd_to_handle operations.

- `int (*prime_handle_to_fd)(struct drm_device *dev, struct drm_file *file_priv, uint32_t handle, uint32_t flags,`

- `int *prime_fd); int (*prime_fd_to_handle)(struct drm_device *dev, struct drm_file *file_priv, int prime_fd, uint32_t *handle);`

- These two operations convert a handle to a PRIME file descriptor and vice versa.
- Drivers must use the kernel dma-buf buffer sharing framework to manage the PRIME file descriptors.
- Non-GEM drivers must implement the operations themselves,
- GEM drivers must use the drm_gem_prime_handle_to_fd and drm_gem_prime_fd_to_handle helper functions.

- Those helpers rely on the driver gem_prime_export and gem_prime_import operations to create a dma-buf instance from a GEM object (dma-buf exporter role) and to create a GEM object from a dma-buf instance (dma-buf importer role).

- `struct dma_buf * (*gem_prime_export)(struct drm_device *dev, struct drm_gem_object *obj, int flags);`
- `struct drm_gem_object * (*gem_prime_import)(struct drm_device *dev, struct dma_buf *dma_buf);`
- These two operations are mandatory for GEM drivers that support DRM PRIME.

DRM PRIME Helper Functions Reference

- Drivers can implement *gem_prime_export* and *gem_prime_import* in terms of simpler APIs by using the helper functions *drm_gem_prime_export* and *drm_gem_prime_import*.
- These functions implement dma-buf support in terms of five lower-level driver callbacks
-

- Export callbacks:
  - *gem_prime_pin* (optional):
    - prepare a GEM object for exporting
  - *gem_prime_get_sg_table*:
    - provide a scatter/gather table of pinned pages
  - *gem_prime_vmap*:
    - vmap a buffer exported by your driver
  - *gem_prime_vunmap*:
    - vunmap a buffer exported by your driver

- Import callback:
  - *gem_prime_import_sg_table* (import):
    - produce a GEM object from another driver's scatter/gather table

GEM Objects Mapping

- Uses the mmap system call on the DRM file handle.
- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
- DRM identifies the GEM object to be mapped by a fake offset passed through the mmap offset argument.
- Prior to being mapped, a GEM object must be associated with a fake offset.
- To do so, drivers must call drm_gem_create_mmap_offset on the object.
-

- The function allocates a fake offset range from a pool and stores the offset divided by PAGE_SIZE in obj->map_list.hash.key.
- Care must be taken not to call drm_gem_create_mmap_offset if a fake offset has already been allocated for the object.
- This can be tested by obj->map_list.map being non-NULL.

- Once allocated, the fake offset value (obj->map_list.hash.key << PAGE_SHIFT) must be passed to the application in a driver-specific way and can then be used as the mmap offset argument.
- The GEM core provides a helper method drm_gem_mmap to handle object mapping.
- The method can be set directly as the mmap file operation handler.
-

- It will look up the GEM object based on the offset value and set the VMA operations to the drm_driver *gem_vm_ops* field.
- drm_gem_mmap doesn't map memory to userspace, but relies on the driver-provided fault handler to map pages individually.

- To use drm_gem_mmap, drivers must fill the struct drm_driver *gem_vm_ops* field with a pointer to VM operations.
- `struct vm_operations_struct *gem_vm_ops`
- `struct vm_operations_struct { void (*open)(struct vm_area_struct * area); void (*close)(struct vm_area_struct * area); int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf); };`

- The open and close operations must update the GEM object reference count.
- Drivers can use the drm_gem_vm_open and drm_gem_vm_close helper functions directly as open and close handlers.
- The fault operation handler is responsible for mapping individual pages to userspace when a page fault occurs.
-

- Depending on the memory allocation scheme, drivers can allocate pages at fault time, or can decide to allocate memory for the GEM object at the time the object is created.
- Drivers that want to map the GEM object upfront instead of handling page faults can implement their own mmap file operation handler.

Dumb GEM Objects

- The GEM API doesn't standardize GEM objects creation and leaves it to driver-specific ioctls.
  - Not an issue for full-fledged graphics stacks that include device-specific userspace components (in libdrm for instance),
  - This limit makes DRM-based early boot graphics unnecessarily complex.
- ≈

- Dumb GEM objects partly alleviate the problem by providing a standard API to create dumb buffers suitable for scanout, which can then be used to create KMS frame buffers.
- To support dumb GEM objects drivers must implement the dumb_create, dumb_destroy and dumb_map_offset operations.
-

- int (*dumb_create)(struct drm_file *file_priv, struct drm_device *dev, struct drm_mode_create_dumb *args);
  - The dumb_create operation creates a GEM object suitable for scanout based on the width, height and depth from the struct drm_mode_create_dumb argument.
  - It fills the argument's *handle*, *pitch* and *size* fields with a handle for the newly created GEM object and its line pitch and size in bytes.

- int (*dumb_destroy)(struct drm_file *file_priv, struct drm_device *dev, uint32_t handle);
  - The dumb_destroy operation destroys a dumb GEM object created by dumb_create.

- `int (*dumb_map_offset)(struct drm_file *file_priv, struct drm_device *dev, uint32_t handle, uint64_t *offset);`
  - The dumb_map_offset operation associates an mmap fake offset with the GEM object given by the handle and returns it.
  - Drivers must use the drm_gem_create_mmap_offset function to associate the fake offset

Memory Coherency

- When mapped to the device or used in a command buffer, backing pages for an object are flushed to memory and marked write combined so as to be coherent with the GPU.
- If the CPU accesses an object after the GPU has finished rendering to the object, then the object must be made coherent with the CPU's view of memory, usually involving GPU cache flushing of various kinds.

- The most important GEM function for GPU devices is providing a command execution interface to clients.
- Client programs construct command buffers containing references to previously allocated memory objects, and then submit them to GEM.

- GEM takes care to bind all the objects into the GTT, execute the buffer, and provide necessary synchronization between clients accessing the same buffers.
- This often involves evicting some objects from the GTT and re-binding others (a fairly expensive operation), and providing relocation support which hides fixed GTT offsets from clients.

- Clients must take care not to submit command buffers that reference more objects than can fit in the GTT;
  - GEM will reject them and no rendering will occur.
  - If several objects in the buffer require fence registers to be allocated for correct rendering (e.g. 2D blits on pre-965 chips), care must be taken not to require more fence registers than are available to the client.
  - Such resource management should be abstracted from the client in libdrm.

Mode Setting

- Drivers must initialize the mode setting core by calling drm_mode_config_init on the DRM device.
- The function initializes the drm_device *mode_config* field and never fails.
- Once done, mode configuration must be setup by initializing the following fields.
- `int min_width, min_height;`
- `int max_width, max_height;`
  - Minimum and maximum width and height of the frame buffers in pixel units.
- `struct drm_mode_config_funcs *funcs;`
  - Mode setting functions.

Frame Buffer Creation

- `struct drm_framebuffer *(*fb_create)(struct drm_device *dev, struct drm_file *file_priv, struct drm_mode_fb_cmd2 *mode_cmd);`
  - Frame buffers are abstract memory objects that provide a source of pixels to scanout to a CRTC.
  - Applications explicitly request the creation of frame buffers through the DRM_IOCTL_MODE_ADDFB(2) ioctls and receive an opaque handle that can be passed to the KMS CRTC control, plane configuration and page flip functions.

- Frame buffers rely on the underneath memory manager for low-level memory operations.
- When creating a frame buffer applications pass a memory handle (or a list of memory handles for multi-planar formats) through the *drm_mode_fb_cmd2* argument.
- Drivers must first validate the requested frame buffer parameters passed through the mode_cmd argument.for sizes, pixel formats or pitches.

- If the parameters are deemed valid, drivers then create, initialize and return an instance of struct drm_framebuffer.
- If desired the instance can be embedded in a larger driver-specific structure.
- Drivers must fill its *width*, *height*, *pitches*, *offsets*, *depth*, *bits_per_pixel* and *pixel_format* fields from the values passed through the *drm_mode_fb_cmd2* argument.

- They should call the drm_helper_mode_fill_fb_struct helper function to do so.
- The initailization of the new framebuffer instance is finalized with a call to drm_framebuffer_init which takes a pointer to DRM frame buffer operations (struct drm_framebuffer_funcs).
- This function publishes the framebuffer that can be accessed concurrently from other threads.
  - Hence it must be the last step in the driver's framebuffer initialization sequence.

- Frame buffer operations are
- int (*create_handle)(struct drm_framebuffer *fb, struct drm_file *file_priv, unsigned int *handle);
  - Create a handle to the frame buffer underlying memory object.
    - If the frame buffer uses a multi-plane format, the handle will reference the memory object associated with the first plane.
- Drivers call drm_gem_handle_create to create the handle.

- **`void (*destroy)(struct drm_framebuffer *framebuffer);`**
  - Destroy the frame buffer object and frees all associated resources.
  - Drivers must call drm_framebuffer_cleanup to free resources allocated by the DRM core for the frame buffer object, and must make sure to unreference all memory objects associated with the frame buffer.
  - Handles created by the create_handle operation are released by the DRM core.

- **`int (*dirty)(struct drm_framebuffer *framebuffer, struct drm_file *file_priv, unsigned flags, unsigned color, struct drm_clip_rect *clips, unsigned num_clips);`**
  - This optional operation notifies the driver that a region of the frame buffer has changed in response to a DRM_IOCTL_MODE_DIRTYFB ioctl call.

- The lifetime of a drm framebuffer is controlled with a reference count, drivers can grab additional references with drm_framebuffer_reference and drop them again with drm_framebuffer_unreference.
- For driver-private framebuffers for which the last reference is never dropped drivers can manually clean up a framebuffer at module unload time with drm_framebuffer_unregister_private.
  - (e.g. for the fbdev framebuffer when the struct drm_framebuffer is embedded into the fbdev helper struct)

Output Polling

- **`void (*output_poll_changed)(struct drm_device *dev);`**
  - This operation notifies the driver that the status of one or more connectors has changed.
  - Drivers that use the fb helper can just call the drm_fb_helper_hotplug_event function to handle this operation.

Locking

- Beside some lookup structures with their own locking (which is hidden behind the interface functions) most of the modeset state is protected by the dev->mode_config.lock mutex and additionally per-crtc locks to allow cursor updates, pageflips and similar operations to occur concurrently with background tasks like output detection.

- Operations which cross domains like a full modeset always grab all locks.
- Drivers there need to protect resources shared between crtcs with additional locking.
- They also need to be careful to always grab the relevant crtc locks if a modset functions touches crtc state
  - e.g. for load detection (which does only grab the mode_config.lock to allow concurrent screen updates on live crtcs).

KMS Initialization and Cleanup

- A KMS device is abstracted and exposed as a set of
  - planes
  - CRTCs
  - encoders
  - connectors.
- KMS drivers must create and initialize all those objects at load time after initializing mode setting.

CRTCs (struct drm_crtc)

- A CRTC is an abstraction representing a part of the chip that contains a pointer to a scanout buffer.
- The number of CRTCs available determines how many independent scanout buffers can be active at any given time.
- The CRTC structure contains
  - a pointer to some video memory (abstracted as a frame buffer object),
  - a display mode, and
  - an (x, y) offset into the video memory to support panning or configurations where one piece of video memory spans multiple CRTCs.

CRTC Initialization

- A KMS device must create and register at least one struct drm_crtc instance.
- The instance is allocated and zeroed by the driver, possibly as part of a larger structure, and registered with a call to drm_crtc_init with a pointer to CRTC functions.

CRTC Operations

- **Set Configuration**
- `int (*set_config)(struct drm_mode_set *set);`
  - Apply a new CRTC configuration to the device.
  - The configuration specifies a CRTC, a frame buffer to scan out from, a (x,y) position in the frame buffer, a display mode and an array of connectors to drive with the CRTC if possible.

- If the frame buffer specified in the configuration is NULL, the driver must detach all encoders connected to the CRTC and all connectors attached to those encoders and disable them.
- This operation is called with the mode config lock held.

Page Flipping

- `int (*page_flip)(struct drm_crtc *crtc, struct drm_framebuffer *fb, struct drm_pending_vblank_event *event);`
  - Schedule a page flip to the given frame buffer for the CRTC. This operation is called with the mode config mutex held.

- Page flipping is a synchronization mechanism that replaces the frame buffer being scanned out by the CRTC with a new frame buffer during vertical blanking, avoiding tearing.
- When an application requests a page flip the DRM core verifies that the new frame buffer is large enough to be scanned out by the CRTC in the currently configured mode and then calls the CRTC page_flip operation with a pointer to the new frame buffer.

- The page_flip operation schedules a page flip.
- Once any pending rendering targeting the new frame buffer has completed, the CRTC will be reprogrammed to display that frame buffer after the next vertical refresh.
- The operation must return immediately without waiting for rendering or page flip to complete and must block any new rendering to the frame buffer until the page flip completes.

- If a page flip can be successfully scheduled the driver must set the drm_crtc->fb field to the new framebuffer pointed to by fb.
  - This is important so that the reference counting on framebuffers stays balanced.
- If a page flip is already pending, the page_flip operation must return -EBUSY.
- To synchronize page flip to vertical blanking the driver will likely need to enable vertical blanking interrupts by calling drm_vblank_get , and call drm_vblank_put after the page flip completes.

- If the application has requested to be notified when page flip completes the page_flip operation will be called with a non-NULL *event* argument pointing to a drm_pending_vblank_event instance.
- Upon page flip completion the driver must call drm_send_vblank_event to fill in the event and send to wake up any waiting processes.

- While waiting for the page flip to complete, the event->base.link list head can be used freely by the driver to store the pending event in a driver-specific list.
- If the file handle is closed before the event is signaled, drivers must take care to destroy the event in theirpreclose operation (and, if needed, call drm_vblank_put).

Miscellaneous

- **`void (*set_property)(struct drm_crtc *crtc, struct drm_property *property, uint64_t value);`**
  - Set the value of the given CRTC property to *value*.
- **`void (*gamma_set)(struct drm_crtc *crtc, u16 *r, u16 *g, u16 *b, uint32_t start, uint32_t size);`**
  - Apply a gamma table to the device. The operation is optional.
- **`void (*destroy)(struct drm_crtc *crtc);`**
  - Destroy the CRTC when not needed anymore

Planes (struct drm_plane)

- A plane represents an image source that can be blended with or overlayed on top of a CRTC during the scanout process.
- Planes are associated with a frame buffer to crop a portion of the image memory (source) and optionally scale it to a destination size.
- The result is then blended with or overlayed on top of a CRTC.

Plane Initialization

- Planes are optional.
- To create a plane, a KMS drivers allocates and zeroes an instance of struct drm_plane(possibly as part of a larger structure) and registers it with a call to drm_plane_init.
- The function takes a bitmask of the CRTCs that can be associated with the plane, a pointer to the plane functions and a list of format supported formats.

Plane Operations

- `int (*update_plane)(struct drm_plane *plane, struct drm_crtc *crtc, struct drm_framebuffer *fb, int crtc_x, int crtc_y, unsigned int crtc_w, unsigned int crtc_h, uint32_t src_x, uint32_t src_y, uint32_t src_w, uint32_t src_h);`
  - Enable and configure the plane to use the given CRTC and frame buffer.
  - The source rectangle in frame buffer memory coordinates is given by the *src_x*, *src_y*, *src_w* and *src_h* parameters (as 16.16 fixed point values).

- Devices that don't support subpixel plane coordinates can ignore the fractional part.
- The destination rectangle in CRTC coordinates is given by
  the *crtc_x*, *crtc_y*, *crtc_w* and *crtc_h*parameters (as integer values).
- Devices scale the source rectangle to the destination rectangle.
- If scaling is not supported, and the source rectangle size doesn't match the destination rectangle size, the driver must return a -EINVAL error.

- **`int (*disable_plane)(struct drm_plane *plane);`**
  - Disable the plane.
  - The DRM core calls this method in response to a DRM_IOCTL_MODE_SETPLANE ioctl call with the frame buffer ID set to 0.
  - Disabled planes must not be processed by the CRTC.
- **`void (*destroy)(struct drm_plane *plane);`**
  - Destroy the plane when not needed anymore.

Encoders (struct drm_encoder)

- An encoder takes pixel data from a CRTC and converts it to a format suitable for any attached connectors.
- On some devices, it may be possible to have a CRTC send data to more than one encoder.
  - In such case, both encoders would receive data from the same scanout buffer, resulting in a "cloned" display configuration across the connectors attached to each encoder.

Encoder Initialization

- A KMS driver must create, initialize and register at least one struct drm_encoder instance.
- The instance is allocated and zeroed by the driver, possibly as part of a larger structure.
- Drivers must initialize the struct drm_encoder *possible_crtcs* and *possible_clones* fields before registering the encoder.
- Both fields are bitmasks of respectively the CRTCs that the encoder can be connected to, and sibling encoders candidate for cloning.

- After being initialized, the encoder must be registered with a call to drm_encoder_init.
- The function takes a pointer to the encoder functions and an encoder type.

- Supported types are
  - DRM_MODE_ENCODER_DAC for VGA and analog on DVI-I/DVI-A
  - DRM_MODE_ENCODER_TMDS for DVI, HDMI and (embedded) DisplayPort
  - DRM_MODE_ENCODER_LVDS for display panels
  - DRM_MODE_ENCODER_TVDAC for TV output (Composite, S-Video, Component, SCART)
  - DRM_MODE_ENCODER_VIRTUAL for virtual machine displays

- Encoders must be attached to a CRTC to be used.
- DRM drivers leave encoders unattached at initialization time.
- Applications (or the fbdev compatibility layer when implemented) are responsible for attaching the encoders they want to use to a CRTC.

Encoder Operations

- **`void (*destroy)(struct drm_encoder *encoder);`**
  - Called to destroy the encoder when not needed anymore.
- **`void (*set_property)(struct drm_plane *plane, struct drm_property *property, uint64_t value);`**
  - Set the value of the given plane property to *value*.

Connectors (struct drm_connector)

- A connector is the final destination for pixel data on a device, and usually connects directly to an external display device like a monitor or laptop panel.
- A connector can only be attached to one encoder at a time.
- The connector is also the structure where information about the attached display is kept, so it contains fields for display data, EDID data, DPMS & connection status, and information about modes supported on the attached displays.

Connector Initialization

- Finally a KMS driver must create, initialize, register and attach at least one struct drm_connector instance.
- The instance is created as other KMS objects and initialized by setting the following fields.
- *interlace_allowed*
  - Whether the connector can handle interlaced modes.
- *doublescan_allowed*
  - Whether the connector can handle doublescan.

- *display_info*
  - Display information is filled from EDID information when a display is detected.
  - For non hot-pluggable displays such as flat panels in embedded systems, the driver should initialize the *display_info.width_mm* and *display_info.height_mm* fields with the physical size of the display.

- *Polled*
  - Connector polling mode, a combination of
  - DRM_CONNECTOR_POLL_HPD
    - The connector generates hotplug events and doesn't need to be periodically polled. The CONNECT and DISCONNECT flags must not be set together with the HPD flag.
  - DRM_CONNECTOR_POLL_CONNECT
    - Periodically poll the connector for connection.
  - DRM_CONNECTOR_POLL_DISCONNEC
    - Periodically poll the connector for disconnection.
  - Set to 0 for connectors that don't support connection status discovery.

- The connector is then registered with a call to drm_connector_init with a pointer to the connector functions and a connector type, and exposed through sysfs with a call to drm_sysfs_connector_add.

- Supported connector types are
  - DRM_MODE_CONNECTOR_VGA
  - DRM_MODE_CONNECTOR_DVII
  - DRM_MODE_CONNECTOR_DVID
  - DRM_MODE_CONNECTOR_DVIA
  - DRM_MODE_CONNECTOR_Composite
  - DRM_MODE_CONNECTOR_SVIDEO

- DRM_MODE_CONNECTOR_LVDS
- DRM_MODE_CONNECTOR_Component
- DRM_MODE_CONNECTOR_9PinDIN
- DRM_MODE_CONNECTOR_DisplayPort
- DRM_MODE_CONNECTOR_HDMIA
- DRM_MODE_CONNECTOR_HDMIB
- DRM_MODE_CONNECTOR_TV
- DRM_MODE_CONNECTOR_eDP
- DRM_MODE_CONNECTOR_VIRTUAL

- Connectors must be attached to an encoder to be used.
- For devices that map connectors to encoders 1:1, the connector should be attached at initialization time with a call to drm_mode_connector_attach_encoder.
- The driver must also set the drm_connector *encoder* field to point to the attached encoder.

- Finally, drivers must initialize the connectors state change detection with a call to drm_kms_helper_poll_init.
- If at least one connector is pollable but can't generate hotplug interrupts (indicated by the DRM_CONNECTOR_POLL_CONNECT and DRM_CONNECTOR_POLL_DISCONNECT connector flags), a delayed work will automatically be queued to periodically poll for changes.

- Connectors that can generate hotplug interrupts must be marked with the DRM_CONNECTOR_POLL_HPD flag instead, and their interrupt handler must call drm_helper_hpd_irq_event.
- The function will queue a delayed work to check the state of all connectors, but no periodic polling will be done.

DPMS

- void (*dpms)(struct drm_connector *connector, int mode);
  - The DPMS operation sets the power state of a connector.
  - The mode argument is one of
    - DRM_MODE_DPMS_ON
    - DRM_MODE_DPMS_STANDBY
    - DRM_MODE_DPMS_SUSPEND
    - DRM_MODE_DPMS_OFF

- In all but DPMS_ON mode the encoder to which the connector is attached should put the display in low-power mode by driving its signals appropriately.
- If more than one connector is attached to the encoder care should be taken not to change the power state of other displays as a side effect.
- Low-power mode should be propagated to the encoders and CRTCs when all related connectors are put in low-power mode.

Modes

- int (*fill_modes)(struct drm_connector *connector, uint32_t max_width, uint32_t max_height);
  - Fill the mode list with all supported modes for the connector.
  - If the *max_width* and *max_height* arguments are non-zero, the implementation must ignore all modes wider than *max_width* or higher than *max_height*.

- The connector must also fill in this operation its *display_info width_mm* and *height_mm* fields with the connected display physical size in millimeters.
- The fields should be set to 0 if the value isn't known or is not applicable (for instance for projector devices).

Connection Status

- The connection status is updated through polling or hotplug events when supported.
- The status value is reported to userspace through ioctls and must not be used inside the driver, as it only gets initialized by a call to drm_mode_getconnector from userspace.
- enum drm_connector_status (*detect)(struct drm_connector *connector, bool force);

- Check to see if anything is attached to the connector.
- The *force* parameter is set to false whilst polling or to true when checking the connector due to user request.
- *force* can be used by the driver to avoid expensive, destructive operations during automated probing.

- Return
  - connector_status_connected
    - if something is connected to the connector,
  - connector_status_disconnected
    - if nothing is connected and
  - connector_status_unknown
    - if the connection state isn't known.

- Drivers should only return connector_status_connected if the connection status has really been probed as connected.
- Connectors that can't detect the connection status, or failed connection status probes, should return connector_status_unknown.

Miscellaneous

- **`void (*set_property)(struct drm_connector *connector, struct drm_property *property, uint64_t value);`**
  - Set the value of the given connector property to *value*.
- **`void (*destroy)(struct drm_connector *connector);`**
  - Destroy the connector when not needed anymore.

Cleanup

- The DRM core manages its objects' lifetime.
- When an object is not needed anymore the core calls its destroy function, which must clean up and free every resource allocated for the object.
- Every drm_*_init call must be matched with a corresponding drm_*_cleanup call to cleanup CRTCs (drm_crtc_cleanup), planes (drm_plane_cleanup), encoders (drm_encoder_cleanup) and connectors (drm_connector_cleanup).

- Connectors that have been added to sysfs must be removed by a call to drm_sysfs_connector_remove before calling drm_connector_cleanup.
- Connectors state change detection must be cleanup up with a call to drm_kms_helper_poll_fini.

Render nodes

- DRM core provides multiple character-devices for user-space to use.
- Depending on which device is opened, user-space can perform a different set of operations (mainly ioctls).
- The primary node is always created and called <term>card<num></term>.

- A currently unused control node, called <term>controlD<num></term> is also created.
- The primary node provides all legacy operations and historically was the only interface used by userspace.
- With KMS, the control node was introduced.

- With the increased use of offscreen renderers and GPGPU applications, clients no longer require running compositors or graphics servers to make use of a GPU.
- DRM API required unprivileged clients to authenticate to a DRM-Master prior to getting GPU access.
- Render nodes were introduced to grant clients GPU access without authentication

- Render nodes solely serve render clients,
  - no modesetting or privileged ioctls can be issued on render nodes.
  - Only non-global rendering commands are allowed.
- If a driver supports render nodes, it must advertise it via the <term>DRIVER_RENDER</term>DRM driver capability.

- If not supported, the primary node must be used for render clients together with the legacy drmAuth authentication procedure.
- If a driver advertises render node support, DRM core will create a separate render node called <term>renderD<num></term>.
- There will be one render node per device.
- No ioctls except PRIME-related ioctls will be allowed on this node.

- Especially <term>GEM_OPEN</term> will be explicitly prohibited.
- Render nodes are designed to avoid the buffer-leaks, which occur if clients guess the flink names or mmap offsets on the legacy interface.

- Drivers must mark their driver-dependent render-only ioctls
  as <term>DRM_RENDER_ALLOW</term> so render clients can use them.
- Drivers should not to allow any privileged ioctls on render nodes.
- With render nodes, user-space can now control access to the render node via basic file-system access-modes

- A running graphics server which authenticates clients on the privileged primary/legacy node is no longer required.
- A client can open the render node and is immediately granted GPU access.
- Communication between clients (or servers) is done via PRIME. FLINK from render node to legacy node is not supported.
- New clients must not use the insecure FLINK interface.

- Besides dropping all modeset/global ioctls, render nodes also drop the DRM-Master concept.
- There is no reason to associate render clients with a DRM-Master as they are independent of any graphics server.
- Besides, they must work without any running master, anyway.

- Drivers must be able to run without a master object if they support render nodes.
- If, on the other hand, a driver requires shared state between clients which is visible to user-space and accessible beyond open-file boundaries, they cannot support render nodes.

Vertical Blanking

- For Tear-free display page flips and/or rendering should be synchronized to vertical blanking.
- The DRM API offers ioctls to perform page flips synchronized to vertical blanking and wait for vertical blanking.
-

- The DRM core handles most of the vertical blanking management logic, which involves
  - filtering out spurious interrupts
  - keeping race-free blanking counters
  - coping with counter wrap-around and resets
  - keeping use counts.
- It relies on the driver to generate vertical blanking interrupts and optionally provide a hardware vertical blanking counter.

- Drivers must implement the following operations.
  - **`int (*enable_vblank) (struct drm_device *dev, int crtc); void (*disable_vblank) (struct drm_device *dev, int crtc);`**
    - Enable or disable vertical blanking interrupts for the given CRTC.
  - **`u32 (*get_vblank_counter) (struct drm_device *dev, int crtc);`**
    - Retrieve the value of the vertical blanking counter for the given CRTC.
    - If the hardware maintains a vertical blanking counter its value should be returned.
    - Otherwise drivers can use thedrm_vblank_count helper function to handle this operation.

- Drivers must initialize the vertical blanking handling core with a call to drm_vblank_init in their loadoperation.
- The function will set the struct drm_device *vblank_disable_allowed* f ield to 0.
- This will keep vertical blanking interrupts enabled permanently until the first mode set operation, where *vblank_disable_allowed* is set to 1.

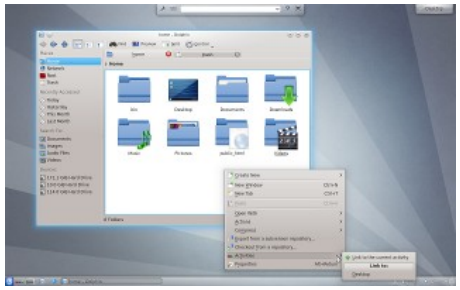
- Drivers can set the field to 1 after calling drm_vblank_init to make vertical blanking interrupts dynamically managed from the beginning.
- Vertical blanking interrupts can be enabled by the DRM core or by drivers themselves (for instance to handle page flipping operations).
- The DRM core maintains a vertical blanking use count to ensure that the interrupts are not disabled while a user still needs them.

- To increment the use count, drivers call drm_vblank_get.
- Upon return vertical blanking interrupts are guaranteed to be enabled.
- To decrement the use count drivers call drm_vblank_put.
- Only when the use count drops to zero will the DRM core disable the vertical blanking interrupts after a delay by scheduling a timer.

- The delay is accessible through the vblankoffdelay module parameter or
  the drm_vblank_offdelay global variable and expressed in milliseconds.
- Its default value is 5000 ms.
- When a vertical blanking interrupt occurs drivers only need to call the drm_handle_vblank function to account for the interrupt.
- Resources allocated by drm_vblank_init must be freed with a call to drm_vblank_cleanup in the driver unload operation handler.

VBlank event handling

- The DRM core exposes two vertical blank related ioctls:
  - DRM_IOCTL_WAIT_VBLANK
    - This takes a struct drm_wait_vblank structure as its argument, and it is used to block or request a signal when a specified vblank event occurs.
  - DRM_IOCTL_MODESET_CTL
    - This should be called by application level drivers before and after mode setting, since on many devices the vertical blank counter is reset at that time.
    - Internally, the DRM snapshots the last vblank count when the ioctl is called with the _DRM_PRE_MODESET command, so that the counter won't go backwards (which is dealt with when _DRM_POST_MODESET is used).


General overview of a modern GPU's functions

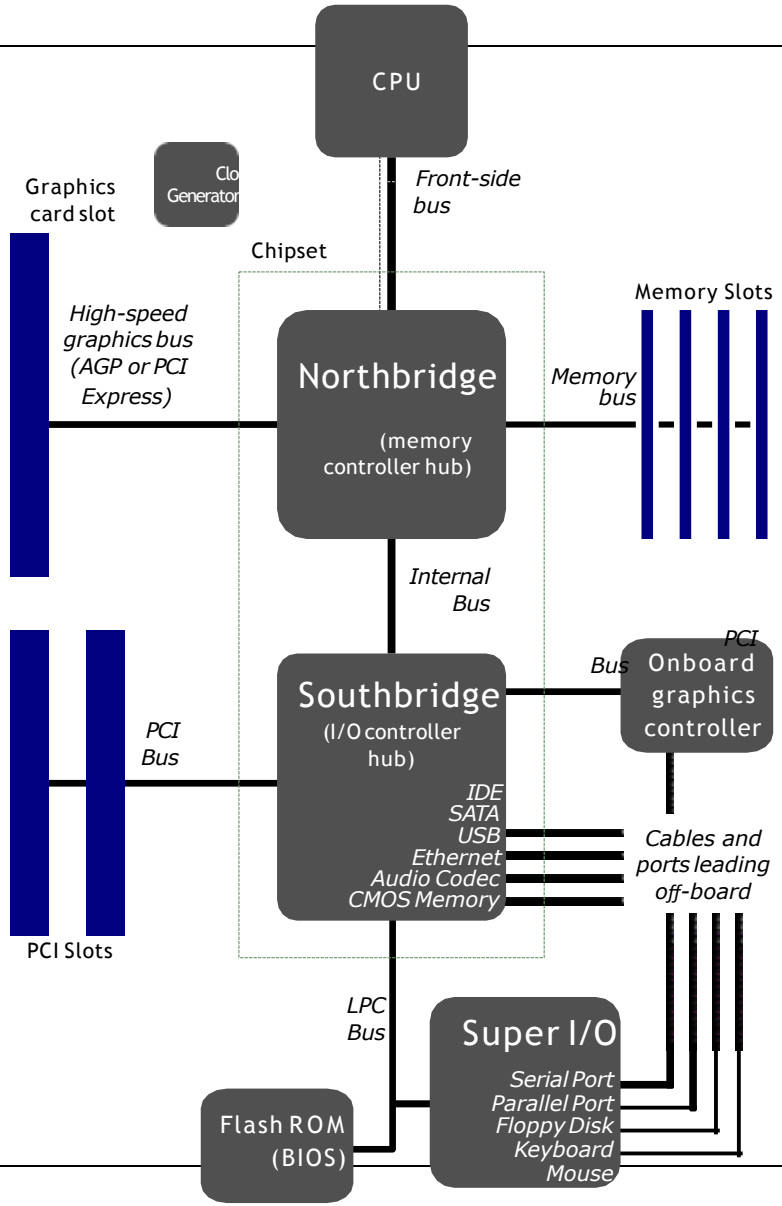
Display content on a screen

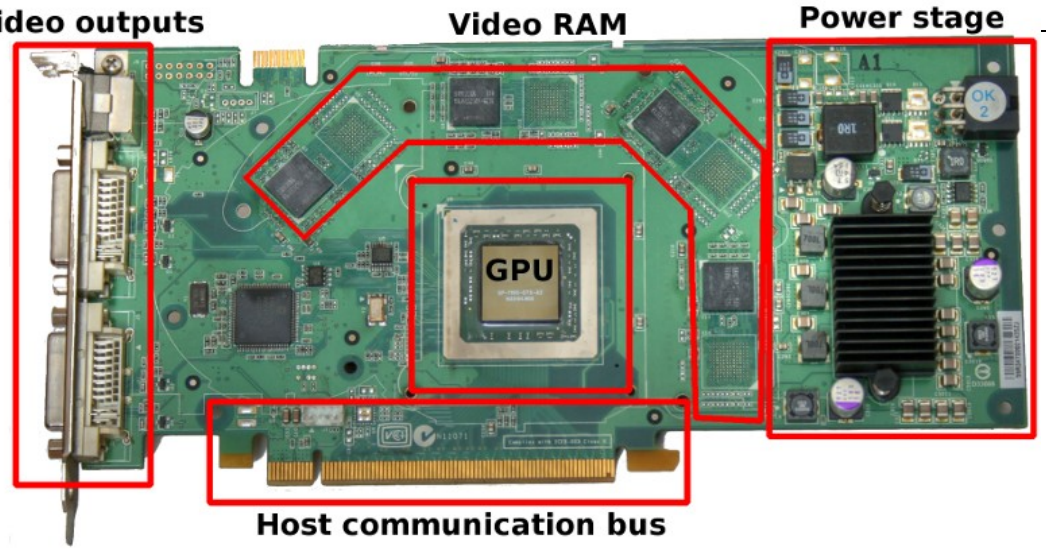
Accelerate 2D operations

Accelerate 3D operations

Decode videos

Accelerate scientific calculations


CPU
Clo
Generator
Graphics
card slot
Front-side
bus
Chipset
Memory Slots
High-speed
graphics bus
(AGP or PCI
Express)
Northbridge
(memory
controller hub)
Memory
bus
Internal
Bus
PCI
Bus
Onboard
graphics
controller
Southbridge
(I/O controller
hub)
PCI
Bus
IDE
SATA
USB
Ethernet
Audio Codec
CMOS Memory
Cables and
ports leading
off-board
PCI Slots
LPC
Bus
Super I/O
Serial Port
Parallel Port
Floppy Disk
Keyboard
Mouse
Flash ROM
(BIOS)



Source: http://www.flickr.com/photos/stefan_ledwina/557505323

Hardware architecture

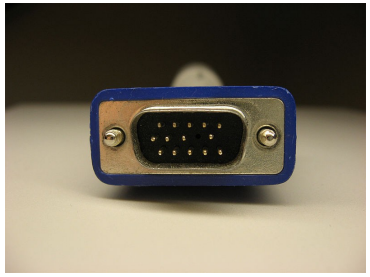
GPU: Where all the calculations are made

VRAM: Stores the textures or general purpose data

Video Outputs: Connects to the screen(s)

Power stage: Lower the voltage, regulate current

Host communication bus: Communication with the CPU




Screen connectors

VGA: Video, introduced in 1987 by IBM

DVI: Video, introduced in 1999 by DDWG

DP: Video & Audio, introduced in 2006 by VESA

HDMI: Video & Audio, introduced in 1999 by HDMI Founders

Example: Some display standards

1981 : Monochrome Display Adapter (MDA)

text-only

monochrome

720 * 350 px or 80*25 characters (50Hz)

1981 : Color Graphics Adapter (CGA)

text & graphics

4 bits (16 colours)

320 * 200 px (60 Hz)

1987 : Video Graphics Array (VGA)

text & graphics

4 bits (16 colours) or 8 bits (256 colours)

320*200px or 640*480px (<= 70 Hz)

Outline

Modern host communication busses

1993 : Peripheral Component Interconnect (PCI)
32 bit & 33.33 MHz
Maximum transfer rate: 133 MB/s

1996 : Accelerated Graphics Port (AGP)
32 bit & 66.66 MHz
Maximum transfer rate: 266 to 2133 MB/s (1x to 8x)

2004 : PCI Express (PCIe)
1 lane: 0.25 − > 2 GB/s (PCIe v1.x − > 4.0)
up to 32 lanes (up to 64 GB/s)
Improve device-to-device communication (no arbitration)

Features

Several generic configuration address spaces (BAR)

Interruption RQuest (IRQ)

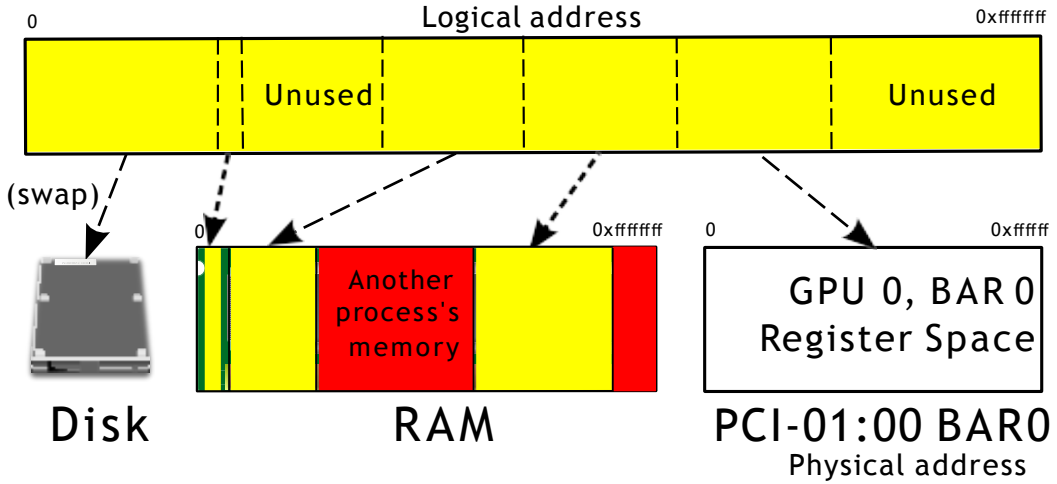
Programming the GPU : Register access via MMIO

A GPU's configuration is mostly stored in registers;

A register is usually identified by an address in a BAR;

We can then access them like memory;

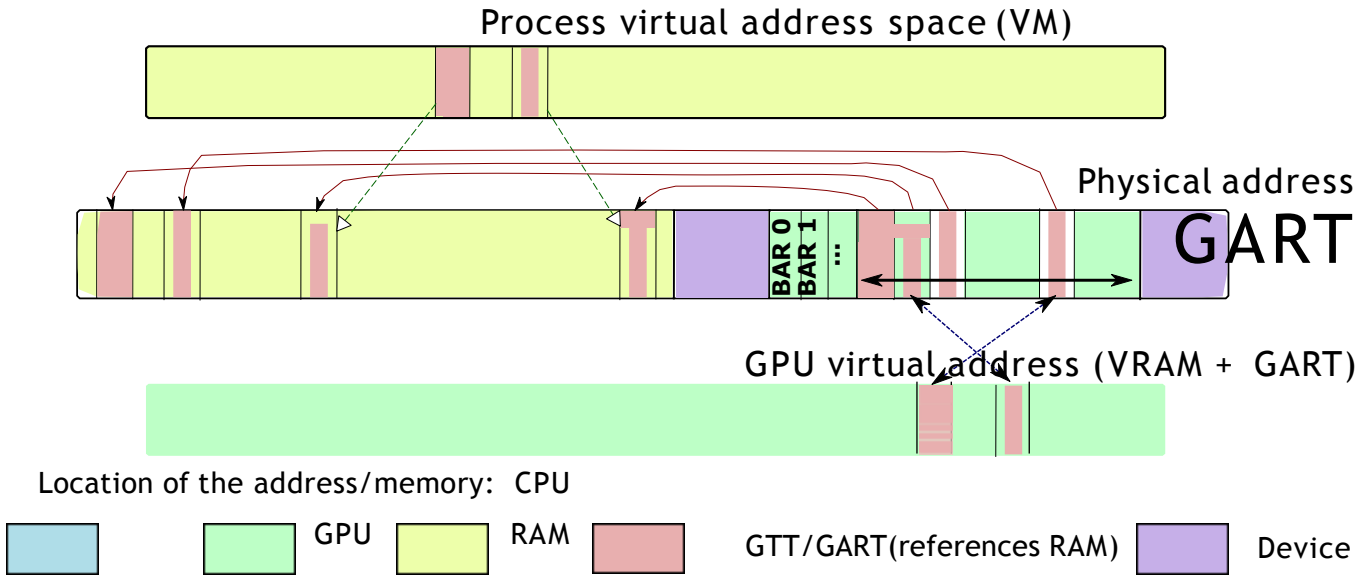
This is called Memory-Mapped Input/Output (MMIO).



Example of a CPU process's virtual memory space

GTT/GART

Providing the GPU with easy access to the Host RAM



GART as a CPU-GPU buffer-sharing mechanism

A program can export buffers to the GPU:

Without actually copying data (faster!);

Allow the GPU to read textures & data from the program;

Outline

The GPU needs the host for:

Setting the screen mode/resolution (mode setting);

Configuring the engines and communication busses;

Handling power management;

Thermal management (fan, react to overheating/power);

Change the GPU's frequencies/voltage to save power;

Processing data:

Allocate processing contexts (GPU VM + context ID);

Upload textures or scientific data;

Send commands to be executed in a context.

Overview of the components of a graphics stack

A GPU with its screen;

One or several input devices (mouse, keyboard);

A windowing system (such as the X-Server and Wayland);

Accelerated-rendering protocols (such as OpenGL);

Graphical applications (such as Firefox or a 3D game).

Components of the Linux Graphics stack

Direct Rendering Manager (DRM) : exports GPU primitives;

X-Server/Wayland : provide a windowing system;

Mesa : provides advanced acceleration APIs;

Outline

Direct Rendering Manager

Inits and configures the GPU; Performs Kernel Mode Setting (KMS); Exports privileged GPU primitives:

Create context + VM allocation;
Command submission;
VRAM memory management: GEM & TTM;
Buffer-sharing: GEM & DMA-Buf;

Implementation is driver-dependent.

libDRM

Wraps the DRM interface into a usable API;

Is meant to be only used by Mesa & the DDX;

Outline

Mesa

Provides advanced acceleration APIs:
3D acceleration: OpenGL / Direct3D
Video acceleration: XVMC, VAAPI, VDPAU

Mostly device-dependent (requires many drivers);

Divided between mesa classics and gallium 3D;

Mesa classics

Old code-base, mostly used by drivers for old cards;

No code sharing between drivers, provide only OpenGL;

Gallium 3D

Built for code-sharing between drivers (State Trackers);
Pipe drivers follow the instructions from the Gallium interface;

Pipe drivers are the device-dependent part of Gallium3D;

Outline

X11 and the X-Server

X11 is a remote rendering API that is 25 years old;

Exports drawing primitives like filled circles, lines;

Is extensible via extensions: eg. DRI2, composite, AIGLX.

The X-Server

Implements the X11 protocol and provides extensions;

Needs a window manager to display windows (like compiz);

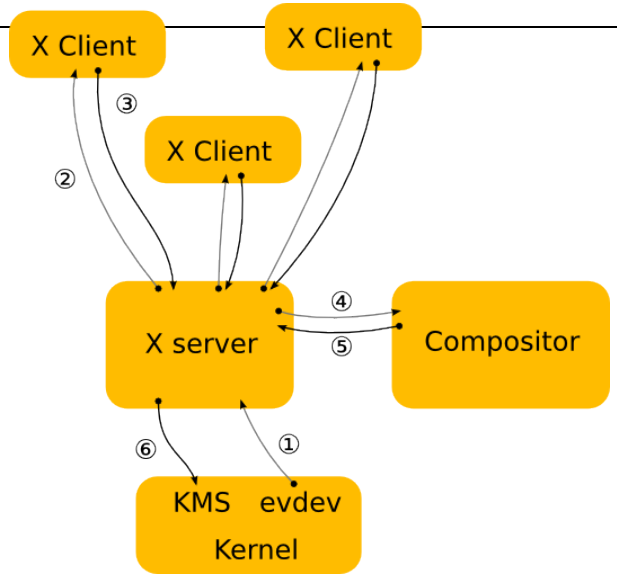
Holds 2D acceleration drivers (DDX): nouveau, radeon, intel;

Logs in /var/log/Xorg.0.log (check them for errors).

The X Resize, Rotate and Reflect Extension (XRandR)

Common X API to configure screens and multi head;

Implemented by the open and proprietary drivers;



Reaction to an input event

1: The kernel driver evdev sends an event to the X-Server; 2: The X-Server forwards it to the window with the focus; 3: The client updates its window and tell the X-Server;

4 & 5: The X-Server lets the compositor update its view; 6: The X-Server sends the new buffer to the GPU.

Outline

Wayland

Protocol started in 2008 by Kristian Høgsberg;

Aims to address some of X11 shortcomings;

Wayland manages:

Input events: Send input events to the right application;

Copy/Paste & Drag'n'Drop;

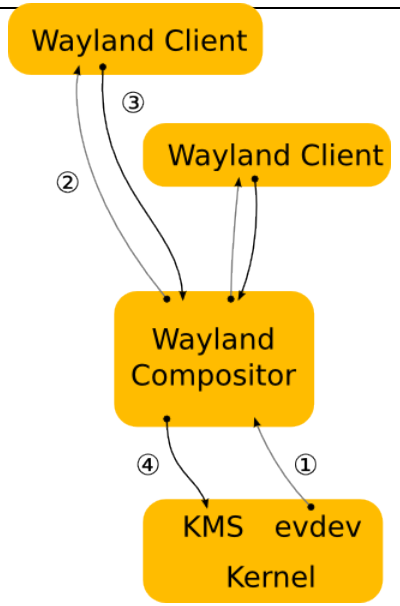
Window buffer sharing (the image representing the window);

Wayland Compositor

Implements the server side of the Wayland protocol;

Talks to Wayland clients and to the driver for compositing;

The reference implementation is called Weston.



Reaction to an input event

1: The kernel driver evdev sends an input event to "Weston"; 2: "Weston" forwards the event to the right Wayland client; 3: The client updates its window and send it to "Weston"; 4: Weston updates its view and send it to the GPU.

Outline

X11 vs Wayland

Rendering protocol vs compositing API:

X11 provides old primitives to get 2D acceleration (such as plain circle, rectangle, ...);

Wayland let applications render their buffers how they want;

Complex & heavy-weight vs minimal & efficient:

X11 is full of old and useless functions that are hard to implement;

Wayland is minimal and only cares about efficient buffer sharing;

Cannot realistically be made secure vs secureable protocol.

X11 : Security

X doesn't care about security and cannot be fixed:
Confidentiality: X applications can spy other applications;
Integrity: X applications can modify other apps' buffers;
Availability: X applications can grab input and be fullscreen.

An X app can get hold of your credentials or bank accounts! An X app can make you believe you are using SSL in Firefox!

Wayland : Security

Wayland is secure if using a secure buffer-sharing mechanism; See https://lwn.net/Articles/517375/.

Outline

Attributions : Anatomy of a GPU

Moxfyre: https://en.wikipedia.org/wiki/File:Motherboard_diagram.svg

Boffy b: https://en.wikipedia.org/wiki/File:IBM_PC_5150.jpg

Katsuki: https://fr.wikipedia.org/wiki/Fichier:VGA_plug.jpg

Evan-Amos: https://fr.wikipedia.org/wiki/Fichier:Dvi-cable.jpg

Evan-Amos: https://en.wikipedia.org/wiki/File:HDMI-Connector.jpg

Andreas -horn- Hornig: https://en.wikipedia.org/wiki/File:Refresh_scan.jpg

Own work: https://en.wikipedia.org/wiki/File:Virtual_memory.svg