

# Linux DRM Developer's Guide

## Jesse Barnes

Initial version  
Intel Corporation

[<jesse.barnes@intel.com>](mailto:jesse.barnes@intel.com)

## Laurent Pinchart

Driver internals  
Ideas on board SPRL

[<laurent.pinchart@ideasonboard.com>](mailto:laurent.pinchart@ideasonboard.com)

Copyright © 2008-2009, 2012 Intel Corporation, Laurent Pinchart

The contents of this file may be used under the terms of the GNU General Public License version 2 (the "GPL") as distributed in the kernel source COPYING file.

Revision History		
Revision 1.0	2012-07-13	LP
Added extensive documentation about driver internals.		

## Table of Contents

- [1. Introduction](#)
- [2. DRM Internals](#)

### [Driver Initialization](#)

- [Driver Information](#)
- [Driver Load](#)

### [Memory management](#)

- [The Translation Table Manager \(TTM\)](#)
- [The Graphics Execution Manager \(GEM\)](#)

### [Mode Setting](#)

- [Frame Buffer Creation](#)
- [Output Polling](#)
- [Locking](#)

### [KMS Initialization and Cleanup](#)

- [CRTCs \(struct drm\\_crtc\)](#)
- [Planes \(struct drm\\_plane\)](#)
- [Encoders \(struct drm\\_encoder\)](#)
- [Connectors \(struct drm\\_connector\)](#)
- [Cleanup](#)
- [Output discovery and initialization example](#)
- [KMS API Functions](#)

### [Mode Setting Helper Functions](#)

- [Helper Functions](#)
- [CRTC Helper Operations](#)
- [Encoder Helper Operations](#)
- [Connector Helper Operations](#)
- [Modeset Helper Functions Reference](#)
- [fbdev Helper Functions Reference](#)
- [Display Port Helper Functions Reference](#)
- [EDID Helper Functions Reference](#)
- [Rectangle Utilities Reference](#)
- [Flip-work Helper Reference](#)
- [VMA Offset Manager](#)

### [KMS Properties](#)

- [Vertical Blanking](#)
- [Open/Close, File Operations and IOCTLs](#)

- [Open and Close](#)
- [File Operations](#)
- [IOCTLs](#)

- Command submission & fencing
- Suspend/Resume
- DMA services

### 3. Userland interfaces

- Render nodes
- VBlank event handling

### A. DRM Driver API

## Chapter 1. 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.

A note on versions: this guide covers features found in the DRM tree, including the TTM memory manager, output configuration and mode setting, and the new vblank internals, in addition to all the regular features found in current kernels.

[Insert diagram of typical DRM stack here]

## Chapter 2. DRM Internals

### Table of Contents

#### Driver Initialization

- Driver Information
- Driver Load

#### Memory management

- The Translation Table Manager (TTM)
- The Graphics Execution Manager (GEM)

#### Mode Setting

- Frame Buffer Creation
- Output Polling
- Locking

#### KMS Initialization and Cleanup

- CRTCs (struct `drm_crtc`)
- Planes (struct `drm_plane`)
- Encoders (struct `drm_encoder`)
- Connectors (struct `drm_connector`)
- Cleanup
- Output discovery and initialization example
- KMS API Functions

#### Mode Setting Helper Functions

- Helper Functions
- CRTC Helper Operations
- Encoder Helper Operations
- Connector Helper Operations
- Modeset Helper Functions Reference
- fbdev Helper Functions Reference
- Display Port Helper Functions Reference
- EDID Helper Functions Reference
- Rectangle Utilities Reference
- Flip-work Helper Reference
- VMA Offset Manager

#### KMS Properties

#### Vertical Blanking

#### Open/Close, File Operations and IOCTLs

- Open and Close
- File Operations
- IOCTLs

#### Command submission & fencing

#### Suspend/Resume

#### DMA services

This chapter documents DRM internals relevant to driver authors and developers working to add support for the latest features to existing drivers.

First, we go over some typical driver initialization requirements, like setting up command buffers, creating an initial output configuration, and initializing core services. Subsequent sections cover core internals in more detail, providing implementation notes and examples.

The DRM layer provides several services to graphics drivers, many of them driven by the application interfaces it provides through `libdrm`, the library

that wraps most of the DRM ioctls. These include vblank event handling, memory management, output management, framebuffer management, command submission & fencing, suspend/resume support, and DMA services.

## Driver Initialization

At the core of every DRM driver is a `drm_driver` structure. Drivers typically statically initialize a `drm_driver` structure, and then pass it to one of the `drm*_init()` functions to register it with the DRM subsystem.

The `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. We will first go through the `drm_driver` static information fields, and will then describe individual operations in details as they get used in later sections.

### Driver Information

#### Driver Features

Drivers inform the DRM core about their requirements and supported features by setting appropriate flags in the `driver_features` field. Since those 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\_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\_IRQ\_SHARED

`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. The installation process is described in [the section called "IRQ Registration"](#).

`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.

### 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 will return an error. Otherwise 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 and otherwise unused by the kernel.

The driver date, formatted as `YYYYMMDD`, is meant to identify the date of the latest modification to the driver. However, as most drivers fail to update it, its value is mostly useless. The DRM core prints it to the kernel log at initialization time and passes it to userspace through the `DRM_IOCTL_VERSION` ioctl.

## 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 (e.g. acquiring clocks, mapping registers or allocating command buffers), initializing the memory manager ([the section called “Memory management”](#)), installing the IRQ handler ([the section called “IRQ Registration”](#)), setting up vertical blanking handling ([the section called “Vertical Blanking”](#)), mode setting ([the section called “Mode Setting”](#)) and initial output configuration ([the section called “KMS Initialization and Cleanup”](#)).

### Note

If compatibility is a concern (e.g. with drivers converted over from User Mode Setting to Kernel Mode Setting), care must be taken to prevent device initialization and control that is incompatible with currently active userspace drivers. For instance, if user level mode setting drivers are in use, it would be problematic to perform output discovery & configuration at load time. Likewise, if user-level drivers unaware of memory management are in use, memory management and command buffer setup may need to be omitted. These requirements are driver-specific, and care needs to be taken to keep both old and new applications and libraries working.

```
int (*load) (struct drm_device *, unsigned long flags);
```

The method takes two arguments, a pointer to the newly created `drm_device` and flags. The flags are used to pass the `driver_data` field of the device id corresponding to the device passed to `drm_*_init()`. Only PCI devices currently use this, USB and platform DRM drivers have their `load` method called with flags to 0.

## 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. This 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.

## IRQ Registration

The DRM core tries to facilitate IRQ handler registration and unregistration by providing `drm_irq_install` and `drm_irq_uninstall` 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

Both the `drm_irq_install` and `drm_irq_uninstall` functions get the device IRQ by calling `drm_dev_to_irq`. This inline function will call 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.

The IRQ will then be 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 thus 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 function calls the optional `irq_postinstall` driver operation. 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 similarly 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 can't 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 Manager Initialization

Every DRM driver requires a memory manager which must be initialized at load time. DRM currently contains two memory managers, the Translation Table Manager (TTM) and the Graphics Execution Manager (GEM). This document describes the use of the GEM memory manager only. See [the section called “Memory management”](#) for details.

## Miscellaneous Device Configuration

Another task that may be necessary for PCI devices during configuration is mapping the video BIOS. On many devices, the VBIOS describes device configuration, LCD panel timings (if any), and contains flags indicating device state. Mapping the BIOS can be done using the `pci_map_rom()` call, a convenience function that takes care of mapping the actual ROM, whether it has been shadowed into memory (typically at address `0xc0000`) or exists on the PCI device in the ROM BAR. Note that after the ROM has been mapped and any necessary information has been extracted, it should be unmapped; on many devices, the ROM address decoder is shared with other BARs, so leaving it mapped could cause undesired behaviour like hangs or memory corruption.

## Memory management

Modern Linux systems require large amount of graphics memory to store frame buffers, textures, vertices and other graphics-related data. Given the very dynamic nature of many of that data, managing graphics memory efficiently is thus crucial for the graphics stack and plays a central role in the DRM infrastructure.

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 that turned out to be hard to use for driver development.

GEM started as an Intel-sponsored project in reaction to TTM's complexity. Its design philosophy is completely different: 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 capabilities and is thus limited to UMA devices.

### The Translation Table Manager (TTM)

TTM design background and information belongs here.

#### TTM initialization

##### Warning

This section is outdated.

Drivers wishing to support TTM must fill out a `drm_bo_driver` structure. The structure contains several fields with function pointers for initializing the TTM, allocating and freeing memory, waiting for command completion and fence synchronization, and memory migration. See the `radeon_ttm.c` file for an example of usage.

The `ttm_global_reference` structure is made up of several fields:

```
struct ttm_global_reference {
    enum ttm_global_types global_type;
    size_t size;
    void *object;
    int (*init) (struct ttm_global_reference *);
    void (*release) (struct ttm_global_reference *);
};
```

There should be one global reference structure for your memory manager as a whole, and there will be others for each object created by the memory manager at runtime. Your global TTM should have a type of `TTM_GLOBAL_TTM_MEM`. The size field for the global object should be `sizeof(struct ttm_mem_global)`, and the init and release hooks should point at your driver-specific init and release routines, which probably eventually call `ttm_mem_global_init` and `ttm_mem_global_release`, respectively.

Once your global TTM accounting structure is set up and initialized by calling `ttm_global_item_ref()` on it, you need to create a buffer object TTM to provide a pool for buffer object allocation by clients and the kernel itself. The type of this object should be `TTM_GLOBAL_TTM_BO`, and its size should be `sizeof(struct ttm_bo_global)`. Again, driver-specific init and release functions may be provided, likely eventually calling `ttm_bo_global_init()` and `ttm_bo_global_release()`, respectively. Also, like the previous object, `ttm_global_item_ref()` is used to create an initial reference count for the TTM, which will call your initialization function.

### The Graphics Execution Manager (GEM)

The GEM design approach has resulted in a memory manager that doesn't provide full coverage of all (or even all common) use cases in its userspace or kernel API. GEM exposes a set of standard memory-related operations to userspace and a set of helper functions to drivers, and let drivers implement hardware-specific operations with their own private API.

The GEM userspace API is described in the [GEM - the Graphics Execution Manager](#) article on LWN. While slightly outdated, the document provides a good overview of the GEM API principles. Buffer allocation and read and write operations, described as part of the common GEM API, are currently implemented using driver-specific ioctls.

GEM is data-agnostic. It manages abstract buffer objects without knowing what individual buffers contain. APIs that require knowledge of buffer contents or purpose, such as buffer allocation or synchronization primitives, are thus outside of the scope of GEM and must be implemented using driver-specific ioctls.

On a fundamental level, GEM involves several operations:

- Memory allocation and freeing
- Command execution
- Aperture management at command execution time

Buffer object allocation is relatively straightforward and largely 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 will then automatically initialize the GEM core before calling the `load` operation. Behind the scene, 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 devices usually have what is called a "stolen" memory region, which provides space for the initial framebuffer and large, contiguous memory regions required by the device. This space is typically 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 usually 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 `shmf`s 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. Note that they can decide to allocate pages when initializing the GEM object, or to 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 is not always desired, for instance when the hardware requires physically contiguous system memory as is often the case in embedded devices. Drivers can create GEM objects with no `shmf`s 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 release by calling `drm_gem_object_reference` and `drm_gem_object_unreference` respectively. The caller must hold the `drm_device` *struct\_mutex* lock. As a convenience, 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. That operation is mandatory for GEM-enabled drivers and must free 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 `shmf`s backing store must be released by calling `drm_gem_object_release` (that function can safely be called if no `shmf`s backing store has been created).

## GEM Objects Naming

Communication between userspace and the kernel refers to GEM objects using local handles, global names or, more recently, 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);

```

Those 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.

While 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

Because mapping operations are fairly heavyweight GEM favours read/write-like access to buffers, implemented through driver-specific ioctls, over mapping buffers to userspace. However, when random access to the buffer is needed (to perform software rendering for instance), direct access to the object can be more efficient.

The `mmap` system call can't be used directly to map GEM objects, as they don't have their own file handle. Two alternative methods currently co-exist to map GEM objects to userspace. The first method uses a driver-specific ioctl to perform the mapping operation, calling `do_mmap` under the hood. This is often considered dubious, seems to be discouraged for new GEM-enabled drivers, and will thus not be described here.

The second method 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 thus 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. Note that `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. While 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 as described in [the section called “GEM Objects Mapping”](#).

## 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. Likewise, 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. This core CPU<->GPU coherency management is provided by a device-specific ioctl, which evaluates an object's current domain and performs any necessary flushing or synchronization to put the object into the desired coherency domain (note that the object may be busy, i.e. an active render target; in that case, setting the domain blocks the client and waits for rendering to complete before performing any necessary flushing operations).

## Command Execution

Perhaps 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. At that point, 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; otherwise, GEM will reject them and no rendering will occur. Similarly, 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. This document assumes that the driver uses GEM, those handles thus reference GEM objects.

Drivers must first validate the requested frame buffer parameters passed through the `mode_cmd` argument. In particular this is where invalid sizes, pixel formats or pitches can be caught.

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 initialization 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`). Note that this function publishes the framebuffer and so from this point on it 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 (e.g. for the fbdev framebuffer when the struct `drm_framebuffer` is embedded into the fbdev helper struct) drivers can manually clean up a framebuffer at module unload time with `drm_framebuffer_unregister_private`.

## 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 modeset 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 and connectors. KMS drivers must thus 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. Therefore, the number of CRTCs available determines how many independent scanout buffers can be active at any given time. The CRTC structure contains several fields to support this: 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.

### Note

FIXME: How should `set_config` interact with DPMS? If the CRTC is suspended, should it be resumed?

### 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. It should call `drm_vblank_get` for that purpose, 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. This can be performed with

```
spin_lock_irqsave(&dev->event_lock, flags);
...
drm_send_vblank_event(dev, pipe, event);
spin_unlock_irqrestore(&dev->event_lock, flags);
```

## Note

FIXME: Could drivers that don't need to wait for rendering to complete just add the event to `dev->vblank_event_list` and let the DRM core handle everything, as for "normal" vertical blanking events?

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 their `preclose` 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*. See [the section called "KMS Properties"](#) for more information about properties.

- `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. See [the section called "KMS Initialization and Cleanup"](#).

## 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 driver 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_SETPANE` 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. See [the section called "KMS Initialization and Cleanup"](#).

## 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 that 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

As for CRTCs, 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. See [the section called "KMS Initialization and Cleanup"](#).

- `void (*set_property)(struct drm_plane *plane,  
                      struct drm_property *property, uint64_t value);`

Set the value of the given plane property to *value*. See [the section called "KMS Properties"](#) for more information about properties.

## 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_DISCONNECT`

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.

## Connector Operations

### Note

Unless otherwise state, all operations are mandatory.

### 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 (see *polled*). The status value is reported to userspace through `ioctl`s 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*. See [the section called "KMS Properties"](#) for more information about properties.

- `void (*destroy)(struct drm_connector *connector);`

Destroy the connector when not needed anymore. See [the section called “KMS Initialization and Cleanup”](#).

## 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 CRTC's (`drm_crtc_cleanup`), planes (`drm_plane_cleanup`), encoders (`drm_encoder_cleanup`) and connectors (`drm_connector_cleanup`). Furthermore, 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 cleaned up with a call to `drm_kms_helper_poll_fini`.

## Output discovery and initialization example

```
void intel_crt_init(struct drm_device *dev)
{
    struct drm_connector *connector;
    struct intel_output *intel_output;

    intel_output = kzalloc(sizeof(struct intel_output), GFP_KERNEL);
    if (!intel_output)
        return;

    connector = &intel_output->base;
    drm_connector_init(dev, &intel_output->base,
                      &intel_crt_connector_funcs, DRM_MODE_CONNECTOR_VGA);

    drm_encoder_init(dev, &intel_output->enc, &intel_crt_enc_funcs,
                    DRM_MODE_ENCODER_DAC);

    drm_mode_connector_attach_encoder(&intel_output->base,
                                     &intel_output->enc);

    /* Set up the DDC bus. */
    intel_output->ddc_bus = intel_i2c_create(dev, GPIOA, "CRTDDC_A");
    if (!intel_output->ddc_bus) {
        dev_printk(KERN_ERR, &dev->pdev->dev, "DDC bus registration "
                  "failed.\n");
        return;
    }

    intel_output->type = INTEL_OUTPUT_ANALOG;
    connector->interlace_allowed = 0;
    connector->doublescan_allowed = 0;

    drm_encoder_helper_add(&intel_output->enc, &intel_crt_helper_funcs);
    drm_connector_helper_add(connector, &intel_crt_connector_helper_funcs);

    drm_sysfs_connector_add(connector);
}
```

In the example above (taken from the i915 driver), a CRTC, connector and encoder combination is created. A device-specific i2c bus is also created for fetching EDID data and performing monitor detection. Once the process is complete, the new connector is registered with sysfs to make its properties available to applications.

## KMS API Functions

### Name

`drm_modeset_lock_all` — take all modeset locks

### Synopsis

```
void fsfuncdrm_modeset_lock_all (dev);
struct drm_device * dev;
```

### Arguments

*dev*  
drm device

### Description

This function takes all modeset locks, suitable where a more fine-grained scheme isn't (yet) implemented.

---

### Name

`drm_modeset_unlock_all` — drop all modeset locks

### Synopsis

```
void fsfuncdrm_modeset_unlock_all (dev);
struct drm_device * dev;
```

## Arguments

*dev*

device

---

## Name

drm\_warn\_on\_modeset\_not\_all\_locked — check that all modeset locks are locked

## Synopsis

```
void fsfuncdrm_warn_on_modeset_not_all_locked (dev);
struct drm_device * dev;
```

## Arguments

*dev*

device

---

## Name

drm\_mode\_object\_find — look up a drm object with static lifetime

## Synopsis

```
struct drm_mode_object * fsfuncdrm_mode_object_find (dev,
                                                    id,
                                                    type);

struct drm_device * dev;
uint32_t id;
uint32_t type;
```

## Arguments

*dev*

drm device

*id*

id of the mode object

*type*

type of the mode object

## Description

Note that framebuffers cannot be looked up with this functions - since those are reference counted, they need special treatment.

---

## Name

drm\_framebuffer\_init — initialize a framebuffer

## Synopsis

```
int fsfuncdrm_framebuffer_init (dev,
                               fb,
                               funcs);

struct drm_device * dev;
struct drm_framebuffer * fb;
const struct drm_framebuffer_funcs * funcs;
```

## Arguments

*dev*

DRM device

*fb*

framebuffer to be initialized

*funcs*

... with these functions

## Description

Allocates an ID for the framebuffer's parent mode object, sets its mode functions & device file and adds it to the master fd list.

## IMPORTANT

This functions publishes the fb and makes it available for concurrent access by other users. Which means by this point the fb `_must_` be fully set up - since all the fb attributes are invariant over its lifetime, no further locking but only correct reference counting is required.

## RETURNS

Zero on success, error code on failure.

---

## Name

`drm_framebuffer_lookup` — look up a drm framebuffer and grab a reference

## Synopsis

```
struct drm_framebuffer * fsfuncdrm_framebuffer_lookup (dev,  
                                                    id);  
  
struct drm_device * dev,  
uint32_t id,
```

## Arguments

*dev*

drm device

*id*

id of the fb object

## Description

If successful, this grabs an additional reference to the framebuffer - callers need to make sure to eventually unreference the returned framebuffer again.

---

## Name

`drm_framebuffer_unreference` — unref a framebuffer

## Synopsis

```
void fsfuncdrm_framebuffer_unreference (fb);  
struct drm_framebuffer * fb,
```

## Arguments

*fb*

framebuffer to unref

## Description

This functions decrements the fb's refcount and frees it if it drops to zero.

---

## Name

`drm_framebuffer_reference` — incr the fb refcnt

## Synopsis

```
void fsfuncdrm_framebuffer_reference (fb);  
struct drm_framebuffer * fb,
```

## Arguments

*fb*  
framebuffer

---

## Name

drm\_framebuffer\_unregister\_private — unregister a private fb from the lookup idr

## Synopsis

```
void fsfuncdrm_framebuffer_unregister_private (fb);  
struct drm_framebuffer * fb;
```

## Arguments

*fb*  
fb to unregister

## Description

Drivers need to call this when cleaning up driver-private framebuffers, e.g. those used for fbdev. Note that the caller must hold a reference of it's own, i.e. the object may not be destroyed through this call (since it'll lead to a locking inversion).

---

## Name

drm\_framebuffer\_cleanup — remove a framebuffer object

## Synopsis

```
void fsfuncdrm_framebuffer_cleanup (fb);  
struct drm_framebuffer * fb;
```

## Arguments

*fb*  
framebuffer to remove

## Description

Cleanup references to a user-created framebuffer. This function is intended to be used from the drivers ->destroy callback.

Note that this function does not remove the fb from active usage - if it is still used anywhere, hilarity can ensue since userspace could call getfb on the id and get back -EINVAL. Obviously no concern at driver unload time.

Also, the framebuffer will not be removed from the lookup idr - for user-created framebuffers this will happen in in the rmfb ioctl. For driver-private objects (e.g. for fbdev) drivers need to explicitly call drm\_framebuffer\_unregister\_private.

---

## Name

drm\_framebuffer\_remove — remove and unreference a framebuffer object

## Synopsis

```
void fsfuncdrm_framebuffer_remove (fb);  
struct drm_framebuffer * fb;
```

## Arguments

*fb*  
framebuffer to remove

## Description

Scans all the CRTCs and planes in *dev*'s mode\_config. If they're using *fb*, removes it, setting it to NULL. Then drops the reference to the passed-in



framebuffer. Might take the modeset locks.

Note that this function optimizes the cleanup away if the caller holds the last reference to the framebuffer. It is also guaranteed to not take the modeset locks in this case.

---

## Name

drm\_crtc\_init — Initialise a new CRTC object

## Synopsis

```
int fsfuncdrm_crtc_init (dev,
                        crtc,
                        funcs);

struct drm_device * dev;
struct drm_crtc * crtc;
const struct drm_crtc_funcs * funcs;
```

## Arguments

*dev*

DRM device

*crtc*

CRTC object to init

*funcs*

callbacks for the new CRTC

## Description

Initiates a new object created as base part of a driver crtc object.

## RETURNS

Zero on success, error code on failure.

---

## Name

drm\_crtc\_cleanup — Clean up the core crtc usage

## Synopsis

```
void fsfuncdrm_crtc_cleanup (crtc);
struct drm_crtc * crtc;
```

## Arguments

*crtc*

CRTC to cleanup

## Description

This function cleans up *crtc* and removes it from the DRM mode setting core. Note that the function does *not* free the crtc structure itself, this is the responsibility of the caller.

---

## Name

drm\_mode\_probed\_add — add a mode to a connector's probed mode list

## Synopsis

```
void fsfuncdrm_mode_probed_add (connector,
                               mode);

struct drm_connector * connector;
struct drm_display_mode * mode;
```

## Arguments

*connector*

connector the new mode

*mode*

mode data

## Description

Add *mode* to *connector*'s mode list for later use.

---

## Name

drm\_connector\_init — Init a preallocated connector

## Synopsis

```
int fsfuncdrm_connector_init (dev,
                             connector,
                             funcs,
                             connector_type);

struct drm_device * dev;
struct drm_connector * connector;
const struct drm_connector_funcs * funcs;
int connector_type;
```

## Arguments

*dev*

DRM device

*connector*

the connector to init

*funcs*

callbacks for this connector

*connector\_type*

user visible type of the connector

## Description

Initialises a preallocated connector. Connectors should be subclassed as part of driver connector objects.

## RETURNS

Zero on success, error code on failure.

---

## Name

drm\_connector\_cleanup — cleans up an initialised connector

## Synopsis

```
void fsfuncdrm_connector_cleanup (connector);
struct drm_connector * connector;
```

## Arguments

*connector*

connector to cleanup

## Description

Cleans up the connector but doesn't free the object.

---

## Name

drm\_plane\_init — Initialise a new plane object

## Synopsis

```
int fsfuncrm_plane_init (dev,
                        plane,
                        possible_crtcs,
                        funcs,
                        formats,
                        format_count,
                        priv);

struct drm_device * dev;
struct drm_plane * plane;
unsigned long possible_crtcs;
const struct drm_plane_funcs * funcs;
const uint32_t * formats;
uint32_t format_count;
bool priv;
```

## Arguments

*dev*

DRM device

*plane*

plane object to init

*possible\_crtcs*

bitmask of possible CRTCs

*funcs*

callbacks for the new plane

*formats*

array of supported formats (DRM\_FORMAT\_\*)

*format\_count*

number of elements in *formats*

*priv*

plane is private (hidden from userspace)?

## Description

Initiates a new object created as base part of a driver plane object.

## RETURNS

Zero on success, error code on failure.

---

## Name

drm\_plane\_cleanup — Clean up the core plane usage

## Synopsis

```
void fsfuncrm_plane_cleanup (plane);
struct drm_plane * plane;
```

## Arguments

*plane*

plane to cleanup

## Description

This function cleans up *plane* and removes it from the DRM mode setting core. Note that the function does *not* free the plane structure itself, this is the responsibility of the caller.

---

## Name

drm\_plane\_force\_disable — Forcibly disable a plane

## Synopsis

```
void fsfuncdrm_plane_force_disable (plane);
struct drm_plane * plane;
```

## Arguments

*plane*

plane to disable

## Description

Forces the plane to be disabled.

Used when the plane's current framebuffer is destroyed, and when restoring fbdev mode.

---

## Name

drm\_mode\_create — create a new display mode

## Synopsis

```
struct drm_display_mode * fsfuncdrm_mode_create (dev);
struct drm_device * dev;
```

## Arguments

*dev*

DRM device

## Description

Create a new `drm_display_mode`, give it an ID, and return it.

## RETURNS

Pointer to new mode on success, NULL on error.

---

## Name

drm\_mode\_destroy — remove a mode

## Synopsis

```
void fsfuncdrm_mode_destroy (dev,
                             mode);
struct drm_device * dev;
struct drm_display_mode * mode;
```

## Arguments

*dev*

DRM device

*mode*

mode to remove

## Description

Free *mode*'s unique identifier, then free it.

---

## Name

drm\_mode\_create\_dvi\_i\_properties — create DVI-I specific connector properties

## Synopsis

```
int fsfuncdrm_mode_create_dvi_i_properties (dev);
struct drm_device * dev;
```

## Arguments

*dev*

DRM device

## Description

Called by a driver the first time a DVI-I connector is made.

---

## Name

drm\_mode\_create\_tv\_properties — create TV specific connector properties

## Synopsis

```
int fsfuncdrm_mode_create_tv_properties (dev,
                                         num_modes,
                                         modes[]);

struct drm_device * dev;
int num_modes;
char * modes[];
```

## Arguments

*dev*

DRM device

*num\_modes*

number of different TV formats (modes) supported

*modes[]*

array of pointers to strings containing name of each format

## Description

Called by a driver's TV initialization routine, this function creates the TV specific connector properties for a given device. Caller is responsible for allocating a list of format names and passing them to this routine.

---

## Name

drm\_mode\_create\_scaling\_mode\_property — create scaling mode property

## Synopsis

```
int fsfuncdrm_mode_create_scaling_mode_property (dev);
struct drm_device * dev;
```

## Arguments

*dev*

DRM device

## Description

Called by a driver the first time it's needed, must be attached to desired connectors.

---

## Name

drm\_mode\_create\_dirty\_info\_property — create dirty property

## Synopsis

```
int fsfuncdrm_mode_create_dirty_info_property (dev);
struct drm_device * dev;
```

## Arguments

*dev*

DRM device

## Description

Called by a driver the first time it's needed, must be attached to desired connectors.

---

## Name

drm\_mode\_set\_config\_internal — helper to call ->set\_config

## Synopsis

```
int fsfuncdrm_mode_set_config_internal (set);
struct drm_mode_set * set;
```

## Arguments

*set*

modeset config to set

## Description

This is a little helper to wrap internal calls to the ->set\_config driver interface. The only thing it adds is correct refcounting dance.

---

## Name

drm\_format\_num\_planes — get the number of planes for format

## Synopsis

```
int fsfuncdrm_format_num_planes (format);
uint32_t format;
```

## Arguments

*format*

pixel format (DRM\_FORMAT\_\*)

## RETURNS

The number of planes used by the specified pixel format.

---

## Name

drm\_format\_plane\_cpp — determine the bytes per pixel value

## Synopsis

```
int fsfuncdrm_format_plane_cpp (format,
                                plane);
uint32_t format;
int plane;
```

## Arguments

*format*

pixel format (DRM\_FORMAT\_\*)

*plane*

plane index

## RETURNS

The bytes per pixel value for the specified plane.

---

### Name

`drm_format_horz_chroma_subsampling` — get the horizontal chroma subsampling factor

## Synopsis

```
int fsfuncdrm_format_horz_chroma_subsampling (format);
uint32_t format;
```

## Arguments

*format*

pixel format (DRM\_FORMAT\_\*)

## RETURNS

The horizontal chroma subsampling factor for the specified pixel format.

---

### Name

`drm_format_vert_chroma_subsampling` — get the vertical chroma subsampling factor

## Synopsis

```
int fsfuncdrm_format_vert_chroma_subsampling (format);
uint32_t format;
```

## Arguments

*format*

pixel format (DRM\_FORMAT\_\*)

## RETURNS

The vertical chroma subsampling factor for the specified pixel format.

---

### Name

`drm_mode_config_init` — initialize DRM mode\_configuration structure

## Synopsis

```
void fsfuncdrm_mode_config_init (dev);
struct drm_device * dev;
```

## Arguments

*dev*

DRM device

## Description

Initialize *dev*'s mode\_config structure, used for tracking the graphics configuration of *dev*.

Since this initializes the modeset locks, no locking is possible. Which is no problem, since this should happen single threaded at init time. It is the driver's problem to ensure this guarantee.

## Name

drm\_mode\_config\_cleanup — free up DRM mode\_config info

## Synopsis

```
void fsfuncdrm_mode_config_cleanup (dev);
struct drm_device * dev;
```

## Arguments

*dev*

DRM device

## Description

Free up all the connectors and CRTCs associated with this DRM device, then free up the framebuffers and associated buffer objects.

Note that since this /should/ happen single-threaded at driver/device teardown time, no locking is required. It's the driver's job to ensure that this guarantee actually holds true.

## FIXME

cleanup any dangling user buffer objects too

## Mode Setting Helper Functions

The CRTC, encoder and connector functions provided by the drivers implement the DRM API. They're called by the DRM core and ioctl handlers to handle device state changes and configuration request. As implementing those functions often requires logic not specific to drivers, mid-layer helper functions are available to avoid duplicating boilerplate code.

The DRM core contains one mid-layer implementation. The mid-layer provides implementations of several CRTC, encoder and connector functions (called from the top of the mid-layer) that pre-process requests and call lower-level functions provided by the driver (at the bottom of the mid-layer). For instance, the `drm_crtc_helper_set_config` function can be used to fill the `struct drm_crtc_funcs` *set\_config* field. When called, it will split the *set\_config* operation in smaller, simpler operations and call the driver to handle them.

To use the mid-layer, drivers call `drm_crtc_helper_add`, `drm_encoder_helper_add` and `drm_connector_helper_add` functions to install their mid-layer bottom operations handlers, and fill the `drm_crtc_funcs`, `drm_encoder_funcs` and `drm_connector_funcs` structures with pointers to the mid-layer top API functions. Installing the mid-layer bottom operation handlers is best done right after registering the corresponding KMS object.

The mid-layer is not split between CRTC, encoder and connector operations. To use it, a driver must provide bottom functions for all of the three KMS entities.

## Helper Functions

- `int drm_crtc_helper_set_config(struct drm_mode_set *set);`

The `drm_crtc_helper_set_config` helper function is a CRTC *set\_config* implementation. It first tries to locate the best encoder for each connector by calling the `connector_best_encoder` helper operation.

After locating the appropriate encoders, the helper function will call the `mode_fixup` encoder and CRTC helper operations to adjust the requested mode, or reject it completely in which case an error will be returned to the application. If the new configuration after mode adjustment is identical to the current configuration the helper function will return without performing any other operation.

If the adjusted mode is identical to the current mode but changes to the frame buffer need to be applied, the `drm_crtc_helper_set_config` function will call the CRTC `mode_set_base` helper operation. If the adjusted mode differs from the current mode, or if the `mode_set_base` helper operation is not provided, the helper function performs a full mode set sequence by calling the `prepare`, `mode_set` and `commit` CRTC and encoder helper operations, in that order.

- `void drm_helper_connector_dpms(struct drm_connector *connector, int mode);`

The `drm_helper_connector_dpms` helper function is a connector *dpms* implementation that tracks power state of connectors. To use the function, drivers must provide *dpms* helper operations for CRTCs and encoders to apply the DPMS state to the device.

The mid-layer doesn't track the power state of CRTCs and encoders. The *dpms* helper operations can thus be called with a mode identical to the currently active mode.

- `int drm_helper_probe_single_connector_modes(struct drm_connector *connector, uint32_t maxX, uint32_t maxY);`

The `drm_helper_probe_single_connector_modes` helper function is a connector *fill\_modes* implementation that updates the connection status for the connector and then retrieves a list of modes by calling the `connector_get_modes` helper operation.

The function filters out modes larger than *max\_width* and *max\_height* if specified. It then calls the `connector_mode_valid` helper operation for each mode in the probed list to check whether the mode is valid for the connector.

## CRTC Helper Operations



- `bool (*mode_fixup)(struct drm_crtc *crtc,  
                    const struct drm_display_mode *mode,  
                    struct drm_display_mode *adjusted_mode);`

Let CRTC's adjust the requested mode or reject it completely. This operation returns true if the mode is accepted (possibly after being adjusted) or false if it is rejected.

The `mode_fixup` operation should reject the mode if it can't reasonably use it. The definition of "reasonable" is currently fuzzy in this context. One possible behaviour would be to set the adjusted mode to the panel timings when a fixed-mode panel is used with hardware capable of scaling. Another behaviour would be to accept any input mode and adjust it to the closest mode supported by the hardware (FIXME: This needs to be clarified).

- `int (*mode_set_base)(struct drm_crtc *crtc, int x, int y,  
                      struct drm_framebuffer *old_fb)`

Move the CRTC on the current frame buffer (stored in `crtc->fb`) to position (x,y). Any of the frame buffer, x position or y position may have been modified.

This helper operation is optional. If not provided, the `drm_crtc_helper_set_config` function will fall back to the `mode_set` helper operation.

### Note

FIXME: Why are x and y passed as arguments, as they can be accessed through `crtc->x` and `crtc->y`?

- `void (*prepare)(struct drm_crtc *crtc);`

Prepare the CRTC for mode setting. This operation is called after validating the requested mode. Drivers use it to perform device-specific operations required before setting the new mode.

- `int (*mode_set)(struct drm_crtc *crtc, struct drm_display_mode *mode,  
                  struct drm_display_mode *adjusted_mode, int x, int y,  
                  struct drm_framebuffer *old_fb);`

Set a new mode, position and frame buffer. Depending on the device requirements, the mode can be stored internally by the driver and applied in the `commit` operation, or programmed to the hardware immediately.

The `mode_set` operation returns 0 on success or a negative error code if an error occurs.

- `void (*commit)(struct drm_crtc *crtc);`

Commit a mode. This operation is called after setting the new mode. Upon return the device must use the new mode and be fully operational.

## Encoder Helper Operations

- `bool (*mode_fixup)(struct drm_encoder *encoder,  
                    const struct drm_display_mode *mode,  
                    struct drm_display_mode *adjusted_mode);`

Let encoders adjust the requested mode or reject it completely. This operation returns true if the mode is accepted (possibly after being adjusted) or false if it is rejected. See the [mode\\_fixup CRTC helper operation](#) for an explanation of the allowed adjustments.

- `void (*prepare)(struct drm_encoder *encoder);`

Prepare the encoder for mode setting. This operation is called after validating the requested mode. Drivers use it to perform device-specific operations required before setting the new mode.

- `void (*mode_set)(struct drm_encoder *encoder,  
                  struct drm_display_mode *mode,  
                  struct drm_display_mode *adjusted_mode);`

Set a new mode. Depending on the device requirements, the mode can be stored internally by the driver and applied in the `commit` operation, or programmed to the hardware immediately.

- `void (*commit)(struct drm_encoder *encoder);`

Commit a mode. This operation is called after setting the new mode. Upon return the device must use the new mode and be fully operational.

## Connector Helper Operations

- `struct drm_encoder *(*best_encoder)(struct drm_connector *connector);`

Return a pointer to the best encoder for the connector. Device that map connectors to encoders 1:1 simply return the pointer to the associated encoder. This operation is mandatory.

- `int (*get_modes)(struct drm_connector *connector);`

Fill the connector's *probed\_modes* list by parsing EDID data with `drm_add_edid_modes` or calling `drm_mode_probed_add` directly for every supported mode and return the number of modes it has detected. This operation is mandatory.

When adding modes manually the driver creates each mode with a call to `drm_mode_create` and must fill the following fields.

- `__u32 type;`

Mode type bitmask, a combination of

`DRM_MODE_TYPE_BUILTIN`

not used?

DRM\_MODE\_TYPE\_CLOCK\_C

not used?

DRM\_MODE\_TYPE\_CRTC\_C

not used?

DRM\_MODE\_TYPE\_PREFERRED - The preferred mode for the connector

not used?

DRM\_MODE\_TYPE\_DEFAULT

not used?

DRM\_MODE\_TYPE\_USERDEF

not used?

DRM\_MODE\_TYPE\_DRIVER

The mode has been created by the driver (as opposed to user-created modes).

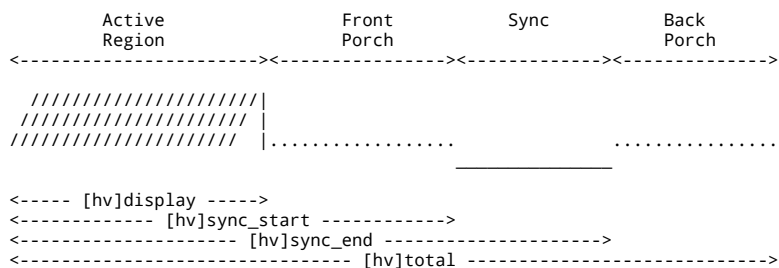
Drivers must set the DRM\_MODE\_TYPE\_DRIVER bit for all modes they create, and set the DRM\_MODE\_TYPE\_PREFERRED bit for the preferred mode.

- `__u32 clock;`

Pixel clock frequency in kHz unit

- `__u16 hdisplay, hsync_start, hsync_end, htotal;`  
`__u16 vdisplay, vsync_start, vsync_end, vtotal;`

Horizontal and vertical timing information



- `__u16 hskew;`  
`__u16 vscan;`

Unknown

- `__u32 flags;`

Mode flags, a combination of

DRM\_MODE\_FLAG\_PHSYNC

Horizontal sync is active high

DRM\_MODE\_FLAG\_NHSYNC

Horizontal sync is active low

DRM\_MODE\_FLAG\_PVSYNC

Vertical sync is active high

DRM\_MODE\_FLAG\_NVSYNC

Vertical sync is active low

DRM\_MODE\_FLAG\_INTERLACE

Mode is interlaced

DRM\_MODE\_FLAG\_DBLSCAN

Mode uses doublescan

DRM\_MODE\_FLAG\_CSYNC

Mode uses composite sync

DRM\_MODE\_FLAG\_PCSYNC

Composite sync is active high

DRM\_MODE\_FLAG\_NCSYNC

Composite sync is active low

DRM\_MODE\_FLAG\_HSKEW

hskew provided (not used?)

DRM\_MODE\_FLAG\_BCAST

not used?

DRM\_MODE\_FLAG\_PIXMUX

not used?

DRM\_MODE\_FLAG\_DBLCLK

not used?

DRM\_MODE\_FLAG\_CLKDIV2

?

Note that modes marked with the INTERLACE or DBLSCAN flags will be filtered out by `drm_helper_probe_single_connector_modes` if the connector's *interlace\_allowed* or *doublescan\_allowed* field is set to 0.

- `char name[DRM_DISPLAY_MODE_LEN];`

Mode name. The driver must call `drm_mode_set_name` to fill the mode name from *hdisplay*, *vdisplay* and interlace flag after filling the corresponding fields.

The *vrefresh* value is computed by `drm_helper_probe_single_connector_modes`.

When parsing EDID data, `drm_add_edid_modes` fill the connector *display\_info width\_mm* and *height\_mm* fields. When creating modes manually the `get_modes` helper operation must set the *display\_info width\_mm* and *height\_mm* fields if they haven't been set already (for instance at initialization time when a fixed-size panel is attached to the connector). The mode *width\_mm* and *height\_mm* fields are only used internally during EDID parsing and should not be set when creating modes manually.

- `int (*mode_valid)(struct drm_connector *connector,  
                  struct drm_display_mode *mode);`

Verify whether a mode is valid for the connector. Return `MODE_OK` for supported modes and one of the enum `drm_mode_status` values (`MODE_*`) for unsupported modes. This operation is mandatory.

As the mode rejection reason is currently not used beside for immediately removing the unsupported mode, an implementation can return `MODE_BAD` regardless of the exact reason why the mode is not valid.

### Note

Note that the `mode_valid` helper operation is only called for modes detected by the device, and *not* for modes set by the user through the CRTC `set_config` operation.

## Modeset Helper Functions Reference

### Name

`drm_helper_move_panel_connectors_to_head` — move panels to the front in the connector list

### Synopsis

```
void fsfuncdrm_helper_move_panel_connectors_to_head (dev);  
struct drm_device * dev;
```

### Arguments

*dev*

drm device to operate on

### Description

Some userspace presumes that the first connected connector is the main display, where it's supposed to display e.g. the login screen. For laptops, this should be the main panel. Use this function to sort all (eDP/LVDS) panels to the front of the connector list, instead of painstakingly trying to initialize them in the right order.

---

### Name

`drm_helper_probe_single_connector_modes` — get complete set of display modes

## Synopsis

```
int fsfuncdrm_helper_probe_single_connector_modes (connector,
                                                    maxX,
                                                    maxY);

struct drm_connector * connector;
uint32_t maxX;
uint32_t maxY;
```

## Arguments

*connector*

connector to probe

*maxX*

max width for modes

*maxY*

max height for modes

## LOCKING

Caller must hold mode config lock.

Based on the helper callbacks implemented by *connector* try to detect all valid modes. Modes will first be added to the connector's probed\_modes list, then culled (based on validity and the *maxX*, *maxY* parameters) and put into the normal modes list.

Intended to be use as a generic implementation of the ->probe *connector* callback for drivers that use the crtc helpers for output mode filtering and detection.

## RETURNS

Number of modes found on *connector*.

---

## Name

drm\_helper\_encoder\_in\_use — check if a given encoder is in use

## Synopsis

```
bool fsfuncdrm_helper_encoder_in_use (encoder);
struct drm_encoder * encoder;
```

## Arguments

*encoder*

encoder to check

## LOCKING

Caller must hold mode config lock.

Walk *encoders*'s DRM device's mode\_config and see if it's in use.

## RETURNS

True if *encoder* is part of the mode\_config, false otherwise.

---

## Name

drm\_helper\_crtc\_in\_use — check if a given CRTC is in a mode\_config

## Synopsis

```
bool fsfuncdrm_helper_crtc_in_use (crtc);
struct drm_crtc * crtc;
```

## Arguments

*crtc*

CRTC to check

## LOCKING

Caller must hold mode config lock.

Walk *crtc*'s DRM device's mode\_config and see if it's in use.

## RETURNS

True if *crtc* is part of the mode\_config, false otherwise.

---

## Name

drm\_helper\_disable\_unused\_functions — disable unused objects

## Synopsis

```
void fsfuncdrm_helper_disable_unused_functions (dev);
struct drm_device * dev;
```

## Arguments

*dev*

DRM device

## LOCKING

Caller must hold mode config lock.

If an connector or CRTC isn't part of *dev*'s mode\_config, it can be disabled by calling its dpms function, which should power it off.

---

## Name

drm\_crtc\_helper\_set\_mode — internal helper to set a mode

## Synopsis

```
bool fsfuncdrm_crtc_helper_set_mode (crtc,
                                     mode,
                                     x,
                                     y,
                                     old_fb);

struct drm_crtc * crtc;
struct drm_display_mode * mode;
int x;
int y;
struct drm_framebuffer * old_fb;
```

## Arguments

*crtc*

CRTC to program

*mode*

mode to use

*x*

horizontal offset into the surface

*y*

vertical offset into the surface

*old\_fb*

old framebuffer, for cleanup

## LOCKING

Caller must hold mode config lock.

Try to set *mode* on *crtc*. Give *crtc* and its associated connectors a chance to fixup or reject the mode prior to trying to set it. This is an internal helper that drivers could e.g. use to update properties that require the entire output pipe to be disabled and re-enabled in a new configuration. For example for changing whether audio is enabled on a hdmi link or for changing panel fitter or dither attributes. It is also called by the `drm_crtc_helper_set_config` helper function to drive the mode setting sequence.

## RETURNS

True if the mode was set successfully, or false otherwise.

---

## Name

`drm_crtc_helper_set_config` — set a new config from userspace

## Synopsis

```
int fsfuncdrm_crtc_helper_set_config (set);
struct drm_mode_set * set;
```

## Arguments

*set*  
mode set configuration

## LOCKING

Caller must hold mode config lock.

Setup a new configuration, provided by the upper layers (either an ioctl call from userspace or internally e.g. from the fbdev support code) in *set*, and enable it. This is the main helper functions for drivers that implement kernel mode setting with the *crtc* helper functions and the assorted *->prepare*, *->modeset* and *->commit* helper callbacks.

## RETURNS

Returns 0 on success, -ERRNO on failure.

---

## Name

`drm_helper_connector_dpms` — connector dpms helper implementation

## Synopsis

```
void fsfuncdrm_helper_connector_dpms (connector,
                                     mode);
struct drm_connector * connector;
int mode;
```

## Arguments

*connector*  
affected connector  
*mode*  
DPMS mode

## Description

This is the main helper function provided by the *crtc* helper framework for implementing the DPMS connector attribute. It computes the new desired DPMS state for all encoders and *crtcs* in the output mesh and calls the *->dpms* callback provided by the driver appropriately.

## fbdev Helper Functions Reference

The fb helper functions are useful to provide an fbdev on top of a drm kernel mode setting driver. They can be used mostly independantly from the *crtc* helper functions used by many drivers to implement the kernel mode setting interfaces.

Initialization is done as a three-step process with `drm_fb_helper_init`, `drm_fb_helper_single_add_all_connectors` and `drm_fb_helper_initial_config`. Drivers with fancier requirements than the default behaviour can override the second step with their own code. Teardown is done with `drm_fb_helper_fini`.

At runtime drivers should restore the fbdev console by calling `drm_fb_helper_restore_fbdev_mode` from their *->lastclose* callback. They should also

notify the fb helper code from updates to the output configuration by calling `drm_fb_helper_hotplug_event`. For easier integration with the output polling code in `drm_crtc_helper.c` the modeset code provides a `->output_poll_changed` callback.

All other functions exported by the fb helper library can be used to implement the fbdev driver interface by the driver.

## Name

`drm_fb_helper_single_add_all_connectors` — add all connectors to fbdev emulation helper

## Synopsis

```
int fsfuncdrm_fb_helper_single_add_all_connectors (fb_helper);
struct drm_fb_helper * fb_helper;
```

## Arguments

*fb\_helper*

fbdev initialized with `drm_fb_helper_init`

## Description

This functions adds all the available connectors for use with the given `fb_helper`. This is a separate step to allow drivers to freely assign connectors to the fbdev, e.g. if some are reserved for special purposes or not adequate to be used for the fbcon.

Since this is part of the initial setup before the fbdev is published, no locking is required.

---

## Name

`drm_fb_helper_debug_enter` — implementation for `->fb_debug_enter`

## Synopsis

```
int fsfuncdrm_fb_helper_debug_enter (info);
struct fb_info * info;
```

## Arguments

*info*

fbdev registered by the helper

---

## Name

`drm_fb_helper_debug_leave` — implementation for `->fb_debug_leave`

## Synopsis

```
int fsfuncdrm_fb_helper_debug_leave (info);
struct fb_info * info;
```

## Arguments

*info*

fbdev registered by the helper

---

## Name

`drm_fb_helper_restore_fbdev_mode` — restore fbdev configuration

## Synopsis

```
bool fsfuncdrm_fb_helper_restore_fbdev_mode (fb_helper);
struct drm_fb_helper * fb_helper;
```

## Arguments

*fb\_helper*

fbcon to restore

## Description

This should be called from driver's drm ->lastclose callback when implementing an fbcon on top of kms using this helper. This ensures that the user isn't greeted with a black screen when e.g. X dies.

---

## Name

drm\_fb\_helper\_blank — implementation for ->fb\_blank

## Synopsis

```
int fsfuncdrm_fb_helper_blank (blank,  
                               info);  
  
int blank;  
struct fb_info * info;
```

## Arguments

*blank*  
desired blanking state

*info*  
fbdev registered by the helper

---

## Name

drm\_fb\_helper\_init — initialize a drm\_fb\_helper structure

## Synopsis

```
int fsfuncdrm_fb_helper_init (dev,  
                             fb_helper,  
                             crtc_count,  
                             max_conn_count);  
  
struct drm_device * dev;  
struct drm_fb_helper * fb_helper;  
int crtc_count;  
int max_conn_count;
```

## Arguments

*dev*  
drm device

*fb\_helper*  
driver-allocated fbdev helper structure to initialize

*crtc\_count*  
maximum number of crtcs to support in this fbdev emulation

*max\_conn\_count*  
max connector count

## Description

This allocates the structures for the fbdev helper with the given limits. Note that this won't yet touch the hardware (through the driver interfaces) nor register the fbdev. This is only done in `drm_fb_helper_initial_config` to allow driver writes more control over the exact init sequence.

Drivers must set `fb_helper->funcs` before calling `drm_fb_helper_initial_config`.

## RETURNS

Zero if everything went ok, nonzero otherwise.

---

## Name



drm\_fb\_helper\_setcmap — implementation for ->fb\_setcmap

## Synopsis

```
int fsfuncdrm_fb_helper_setcmap (cmap,  
                                info);  
  
struct fb_cmap * cmap;  
struct fb_info * info;
```

## Arguments

*cmap*

cmap to set

*info*

fbdev registered by the helper

---

## Name

drm\_fb\_helper\_check\_var — implementation for ->fb\_check\_var

## Synopsis

```
int fsfuncdrm_fb_helper_check_var (var,  
                                   info);  
  
struct fb_var_screeninfo * var;  
struct fb_info * info;
```

## Arguments

*var*

screeninfo to check

*info*

fbdev registered by the helper

---

## Name

drm\_fb\_helper\_set\_par — implementation for ->fb\_set\_par

## Synopsis

```
int fsfuncdrm_fb_helper_set_par (info);  
struct fb_info * info;
```

## Arguments

*info*

fbdev registered by the helper

## Description

This will let fbcon do the mode init and is called at initialization time by the fbdev core when registering the driver, and later on through the hotplug callback.

---

## Name

drm\_fb\_helper\_pan\_display — implementation for ->fb\_pan\_display

## Synopsis

```
int fsfuncdrm_fb_helper_pan_display (var,  
                                     info);  
  
struct fb_var_screeninfo * var;  
struct fb_info * info;
```

# Arguments

<i>var</i>	updated screen information
<i>info</i>	fbdev registered by the helper

---

# Name

drm\_fb\_helper\_fill\_fix — initializes fixed fbdev information

# Synopsis

```
void fsfuncdrm_fb_helper_fill_fix (info,  
                                  pitch,  
                                  depth);  
  
struct fb_info * info;  
uint32_t pitch;  
uint32_t depth;
```

# Arguments

<i>info</i>	fbdev registered by the helper
<i>pitch</i>	desired pitch
<i>depth</i>	desired depth

# Description

Helper to fill in the fixed fbdev information useful for a non-accelerated fbdev emulations. Drivers which support acceleration methods which impose additional constraints need to set up their own limits.

Drivers should call this (or their equivalent setup code) from their ->fb\_probe callback.

---

# Name

drm\_fb\_helper\_fill\_var — initalizes variable fbdev information

# Synopsis

```
void fsfuncdrm_fb_helper_fill_var (info,  
                                  fb_helper,  
                                  fb_width,  
                                  fb_height);  
  
struct fb_info * info;  
struct drm_fb_helper * fb_helper;  
uint32_t fb_width;  
uint32_t fb_height;
```

# Arguments

<i>info</i>	fbdev instance to set up
<i>fb_helper</i>	fb helper instance to use as template
<i>fb_width</i>	desired fb width
<i>fb_height</i>	desired fb height

## Description

Sets up the variable fbdev metainformation from the given fb helper instance and the drm framebuffer allocated in fb\_helper->fb.

Drivers should call this (or their equivalent setup code) from their ->fb\_probe callback after having allocated the fbdev backing storage framebuffer.

---

## Name

drm\_fb\_helper\_initial\_config — setup a sane initial connector configuration

## Synopsis

```
bool fsfuncdrm_fb_helper_initial_config (fb_helper,
                                         bpp_sel);

struct drm_fb_helper * fb_helper;
int bpp_sel;
```

## Arguments

*fb\_helper*

fb\_helper device struct

*bpp\_sel*

bpp value to use for the framebuffer configuration

## Description

Scans the CRTCs and connectors and tries to put together an initial setup. At the moment, this is a cloned configuration across all heads with a new framebuffer object as the backing store.

Note that this also registers the fbdev and so allows userspace to call into the driver through the fbdev interfaces.

This function will call down into the ->fb\_probe callback to let the driver allocate and initialize the fbdev info structure and the drm framebuffer used to back the fbdev. drm\_fb\_helper\_fill\_var and drm\_fb\_helper\_fill\_fix are provided as helpers to setup simple default values for the fbdev info structure.

## RETURNS

Zero if everything went ok, nonzero otherwise.

---

## Name

drm\_fb\_helper\_hotplug\_event — respond to a hotplug notification by probing all the outputs attached to the fb

## Synopsis

```
int fsfuncdrm_fb_helper_hotplug_event (fb_helper);
struct drm_fb_helper * fb_helper;
```

## Arguments

*fb\_helper*

the drm\_fb\_helper

## Description

Scan the connectors attached to the fb\_helper and try to put together a setup after \*notification of a change in output configuration.

Called at runtime, takes the mode config locks to be able to check/change the modeset configuration. Must be run from process context (which usually means either the output polling work or a work item launched from the driver's hotplug interrupt).

Note that the driver must ensure that this is only called \_after\_ the fb has been fully set up, i.e. after the call to drm\_fb\_helper\_initial\_config.

## RETURNS

0 on success and a non-zero error code otherwise.

---

## Name

struct drm\_fb\_helper\_funcs — driver callbacks for the fbdev emulation library

## Synopsis

```
struct drm_fb_helper_funcs {
    void (* gamma_set) (struct drm_crtc *crtc, u16 red, u16 green,u16 blue, int regno);
    void (* gamma_get) (struct drm_crtc *crtc, u16 *red, u16 *green,u16 *blue, int regno);
    int (* fb_probe) (struct drm_fb_helper *helper,struct drm_fb_helper_surface_size *sizes);
    bool (* initial_config) (struct drm_fb_helper *fb_helper,struct drm_fb_helper_crtc **crtcs,struct drm_display_mode **modes,bool *enable
};
```

## Members

gamma\_set

Set the given gamma lut register on the given crtc.

gamma\_get

Read the given gamma lut register on the given crtc, used to save the current lut when force-restoring the fbdev for e.g. kdbg.

fb\_probe

Driver callback to allocate and initialize the fbdev info structure. Futhermore it also needs to allocate the drm framebuffer used to back the fbdev.

initial\_config

Setup an initial fbdev display configuration

## Description

Driver callbacks used by the fbdev emulation helper library.

### Display Port Helper Functions Reference

These functions contain some common logic and helpers at various abstraction levels to deal with Display Port sink devices and related things like DP aux channel transfers, EDID reading over DP aux channels, decoding certain DPCD blocks, ...

## Name

struct i2c\_algo\_dp\_aux\_data — driver interface structure for i2c over dp aux algorithm

## Synopsis

```
struct i2c_algo_dp_aux_data {
    bool running;
    u16 address;
    int (* aux_ch) (struct i2c_adapter *adapter,int mode, uint8_t write_byte,uint8_t *read_byte);
};
```

## Members

running

set by the algo indicating whether an i2c is ongoing or whether the i2c bus is quiescent

address

i2c target address for the currently ongoing transfer

aux\_ch

driver callback to transfer a single byte of the i2c payload

---

## Name

i2c\_dp\_aux\_add\_bus — register an i2c adapter using the aux ch helper

## Synopsis

```
int fsfunci2c_dp_aux_add_bus (adapter);
struct i2c_adapter * adapter;
```

## Arguments

*adapter*

i2c adapter to register

## Description

This registers an i2c adapater that uses dp aux channel as it's underlaying transport. The driver needs to fill out the `i2c_algo_dp_aux_data` structure and store it in the `algo_data` member of the *adapter* argument. This will be used by the i2c over dp aux algorithm to drive the hardware.

## RETURNS

0 on success, -ERRNO on failure.

### EDID Helper Functions Reference

## Name

`drm_edid_is_valid` — sanity check EDID data

## Synopsis

```
bool fsfuncdrm_edid_is_valid (edid);
struct edid * edid;
```

## Arguments

*edid*  
EDID data

## Description

Sanity-check an entire EDID record (including extensions)

---

## Name

`drm_probe_ddc` —

## Synopsis

```
bool fsfuncdrm_probe_ddc (adapter);
struct i2c_adapter * adapter;
```

## Arguments

*adapter*  
-- undescribed --

## Description

\param adapter : i2c device adaptor \return 1 on success

---

## Name

`drm_get_edid` — get EDID data, if available

## Synopsis

```
struct edid * fsfuncdrm_get_edid (connector,
                                  adapter);
struct drm_connector * connector;
struct i2c_adapter * adapter;
```

## Arguments

*connector*  
connector we're probing  
*adapter*  
i2c adapter to use for DDC

## Description

Poke the given i2c channel to grab EDID data if possible. If found, attach it to the connector.

Return edid data or NULL if we couldn't find any.

---

## Name

drm\_match\_cea\_mode — look for a CEA mode matching given mode

## Synopsis

```
u8 fsfuncdrm_match_cea_mode (to_match);
const struct drm_display_mode * to_match;
```

## Arguments

*to\_match*

display mode

## Description

Returns the CEA Video ID (VIC) of the mode or 0 if it isn't a CEA-861 mode.

---

## Name

drm\_edid\_to\_eld — build ELD from EDID

## Synopsis

```
void fsfuncdrm_edid_to_eld (connector,
                           edid);
struct drm_connector * connector;
struct edid * edid;
```

## Arguments

*connector*

connector corresponding to the HDMI/DP sink

*edid*

EDID to parse

## Description

Fill the ELD (EDID-Like Data) buffer for passing to the audio driver.

## Some ELD fields are left to the graphics driver caller

- Conn\_Type - HDCP - Port\_ID

---

## Name

drm\_edid\_to\_sad — extracts SADs from EDID

## Synopsis

```
int fsfuncdrm_edid_to_sad (edid,
                          sads);
struct edid * edid;
struct cea_sad ** sads;
```

## Arguments

*edid*

EDID to parse

*sads*

pointer that will be set to the extracted SADs

# Description

Looks for CEA EDID block and extracts SADs (Short Audio Descriptors) from it.

## Note

returned pointer needs to be kfree

Return number of found SADs or negative number on error.

---

## Name

drm\_edid\_to\_speaker\_allocation — extracts Speaker Allocation Data Blocks from EDID

## Synopsis

```
int fsfuncrm_edid_to_speaker_allocation (edid,
                                         sadb);

struct edid * edid;
u8 ** sadb;
```

## Arguments

- edid*  
EDID to parse
- sadb*  
pointer to the speaker block

## Description

Looks for CEA EDID block and extracts the Speaker Allocation Data Block from it.

## Note

returned pointer needs to be kfree

Return number of found Speaker Allocation Blocks or negative number on error.

---

## Name

drm\_av\_sync\_delay — HDMI/DP sink audio-video sync delay in millisecond

## Synopsis

```
int fsfuncrm_av_sync_delay (connector,
                             mode);

struct drm_connector * connector;
struct drm_display_mode * mode;
```

## Arguments

- connector*  
connector associated with the HDMI/DP sink
- mode*  
the display mode

---

## Name

drm\_select\_eld — select one ELD from multiple HDMI/DP sinks

## Synopsis

```
struct drm_connector * fsfuncrm_select_eld (encoder,
                                             mode);

struct drm_encoder * encoder;
struct drm_display_mode * mode;
```

## Arguments

*encoder*

the encoder just changed display mode

*mode*

the adjusted display mode

## Description

It's possible for one encoder to be associated with multiple HDMI/DP sinks. The policy is now hard coded to simply use the first HDMI/DP sink's ELD.

---

## Name

drm\_detect\_hdmi\_monitor — detect whether monitor is hdmi.

## Synopsis

```
bool fsfuncdrm_detect_hdmi_monitor (edid);
struct edid * edid,
```

## Arguments

*edid*

monitor EDID information

## Description

Parse the CEA extension according to CEA-861-B. Return true if HDMI, false if not or unknown.

---

## Name

drm\_detect\_monitor\_audio — check monitor audio capability

## Synopsis

```
bool fsfuncdrm_detect_monitor_audio (edid);
struct edid * edid,
```

## Arguments

*edid*

-- undescribed --

## Description

Monitor should have CEA extension block. If monitor has 'basic audio', but no CEA audio blocks, it's 'basic audio' only. If there is any audio extension block and supported audio format, assume at least 'basic audio' support, even if 'basic audio' is not defined in EDID.

---

## Name

drm\_rgb\_quant\_range\_selectable — is RGB quantization range selectable?

## Synopsis

```
bool fsfuncdrm_rgb_quant_range_selectable (edid);
struct edid * edid,
```

## Arguments

*edid*

-- undescribed --



## Description

Check whether the monitor reports the RGB quantization range selection as supported. The AVI infoframe can then be used to inform the monitor which quantization range (full or limited) is used.

---

## Name

drm\_add\_edid\_modes — add modes from EDID data, if available

## Synopsis

```
int fsfuncrm_add_edid_modes (connector,  
                             edid);  
struct drm_connector * connector;  
struct edid * edid;
```

## Arguments

*connector*

connector we're probing

*edid*

edid data

## Description

Add the specified modes to the connector's mode list.

Return number of modes added or 0 if we couldn't find any.

---

## Name

drm\_add\_modes\_noedid — add modes for the connectors without EDID

## Synopsis

```
int fsfuncrm_add_modes_noedid (connector,  
                               hdisplay,  
                               vdisplay);  
struct drm_connector * connector;  
int hdisplay;  
int vdisplay;
```

## Arguments

*connector*

connector we're probing

*hdisplay*

the horizontal display limit

*vdisplay*

the vertical display limit

## Description

Add the specified modes to the connector's mode list. Only when the hdisplay/vdisplay is not beyond the given limit, it will be added.

Return number of modes added or 0 if we couldn't find any.

---

## Name

drm\_hdmi\_avi\_infoframe\_from\_display\_mode — fill an HDMI AVI infoframe with data from a DRM display mode

## Synopsis

```
int fsfuncrm_hdmi_avi_infoframe_from_display_mode (frame,  
                                                    mode);  
struct hdmi_avi_infoframe * frame;
```

```
const struct drm_display_mode * mode;
```

Arguments

<i>frame</i>	HDMI AVI infoframe
<i>mode</i>	DRM display mode

Description

Returns 0 on success or a negative error code on failure.

Name

drm\_hdmi\_vendor\_infoframe\_from\_display\_mode — fill an HDMI infoframe with data from a DRM display mode

Synopsis

```
int fsfuncdrm_hdmi_vendor_infoframe_from_display_mode (frame,  
                                                       mode);  
  
struct hdmi_vendor_infoframe * frame;  
const struct drm_display_mode * mode;
```

Arguments

<i>frame</i>	HDMI vendor infoframe
<i>mode</i>	DRM display mode

Description

Note that there's is a need to send HDMI vendor infoframes only when using a 4k or stereoscopic 3D mode. So when giving any other mode as input this function will return -EINVAL, error that can be safely ignored.

Returns 0 on success or a negative error code on failure.

Rectangle Utilities Reference

Utility functions to help manage rectangular areas for clipping, scaling, etc. calculations.

Name

struct drm\_rect — two dimensional rectangle

Synopsis

```
struct drm_rect {  
    int x1;  
    int y1;  
    int x2;  
    int y2;  
};
```

Members

x1	horizontal starting coordinate (inclusive)
y1	vertical starting coordinate (inclusive)
x2	horizontal ending coordinate (exclusive)
y2	vertical ending coordinate (exclusive)



```
int horz;
int vert;
```

Arguments

- r*  
rectangle to be downscaled
- horz*  
horizontal downscale factor
- vert*  
vertical downscale factor

Description

Divide the coordinates of rectangle *r* by *horz* and *vert*.

---

Name

drm\_rect\_width — determine the rectangle width

Synopsis

```
int fsfuncdrm_rect_width (r);
const struct drm_rect * r;
```

Arguments

- r*  
rectangle whose width is returned

RETURNS

The width of the rectangle.

---

Name

drm\_rect\_height — determine the rectangle height

Synopsis

```
int fsfuncdrm_rect_height (r);
const struct drm_rect * r;
```

Arguments

- r*  
rectangle whose height is returned

RETURNS

The height of the rectangle.

---

Name

drm\_rect\_visible — determine if the the rectangle is visible

Synopsis

```
bool fsfuncdrm_rect_visible (r);
const struct drm_rect * r;
```

Arguments

*r*

rectangle whose visibility is returned

## RETURNS

true if the rectangle is visible, false otherwise.

---

## Name

drm\_rect\_equals — determine if two rectangles are equal

## Synopsis

```
bool fsfuncdrm_rect_equals (r1,  
                             r2);  
const struct drm_rect * r1;  
const struct drm_rect * r2;
```

## Arguments

*r1*

first rectangle

*r2*

second rectangle

## RETURNS

true if the rectangles are equal, false otherwise.

---

## Name

drm\_rect\_intersect — intersect two rectangles

## Synopsis

```
bool fsfuncdrm_rect_intersect (r1,  
                               r2);  
struct drm_rect * r1;  
const struct drm_rect * r2;
```

## Arguments

*r1*

first rectangle

*r2*

second rectangle

## Description

Calculate the intersection of rectangles *r1* and *r2*. *r1* will be overwritten with the intersection.

## RETURNS

true if rectangle *r1* is still visible after the operation, false otherwise.

---

## Name

drm\_rect\_clip\_scaled — perform a scaled clip operation

## Synopsis

```
bool fsfuncdrm_rect_clip_scaled (src,  
                                 dst,  
                                 clip,  
                                 hscale,
```

```
                                vscale);
struct drm_rect * src;
struct drm_rect * dst;
const struct drm_rect * clip;
int hscale;
int vscale;
```

Arguments

- src
source window rectangle
- dst
destination window rectangle
- clip
clip rectangle
- hscale
horizontal scaling factor
- vscale
vertical scaling factor

Description

Clip rectangle dst by rectangle clip. Clip rectangle src by the same amounts multiplied by hscale and vscale.

RETURNS

true if rectangle dst is still visible after being clipped, false otherwise

Name

drm\_rect\_calc\_hscale — calculate the horizontal scaling factor

Synopsis

```
int fsfuncdrm_rect_calc_hscale (src,
                                dst,
                                min_hscale,
                                max_hscale);

const struct drm_rect * src;
const struct drm_rect * dst;
int min_hscale;
int max_hscale;
```

Arguments

- src
source window rectangle
- dst
destination window rectangle
- min\_hscale
minimum allowed horizontal scaling factor
- max\_hscale
maximum allowed horizontal scaling factor

Description

Calculate the horizontal scaling factor as (src width) / (dst width).

RETURNS

The horizontal scaling factor, or errno of out of limits.

## Name

drm\_rect\_calc\_vscale — calculate the vertical scaling factor

## Synopsis

```
int fsfuncdrm_rect_calc_vscale (src,
                                dst,
                                min_vscale,
                                max_vscale);

const struct drm_rect * src;
const struct drm_rect * dst;
int min_vscale;
int max_vscale;
```

## Arguments

*src*

source window rectangle

*dst*

destination window rectangle

*min\_vscale*

minimum allowed vertical scaling factor

*max\_vscale*

maximum allowed vertical scaling factor

## Description

Calculate the vertical scaling factor as (*src* height) / (*dst* height).

## RETURNS

The vertical scaling factor, or errno of out of limits.

---

## Name

drm\_rect\_calc\_hscale\_relaxed — calculate the horizontal scaling factor

## Synopsis

```
int fsfuncdrm_rect_calc_hscale_relaxed (src,
                                         dst,
                                         min_hscale,
                                         max_hscale);

struct drm_rect * src;
struct drm_rect * dst;
int min_hscale;
int max_hscale;
```

## Arguments

*src*

source window rectangle

*dst*

destination window rectangle

*min\_hscale*

minimum allowed horizontal scaling factor

*max\_hscale*

maximum allowed horizontal scaling factor

## Description

Calculate the horizontal scaling factor as (*src* width) / (*dst* width).

If the calculated scaling factor is below *min\_vscale*, decrease the height of rectangle *dst* to compensate.

If the calculated scaling factor is above *max\_vscale*, decrease the height of rectangle *src* to compensate.

## RETURNS

The horizontal scaling factor.

---

## Name

drm\_rect\_calc\_vscale\_relaxed — calculate the vertical scaling factor

## Synopsis

```
int fsfuncdrm_rect_calc_vscale_relaxed (src,
                                       dst,
                                       min_vscale,
                                       max_vscale);

struct drm_rect * src;
struct drm_rect * dst;
int min_vscale;
int max_vscale;
```

## Arguments

- src*  
source window rectangle
- dst*  
destination window rectangle
- min\_vscale*  
minimum allowed vertical scaling factor
- max\_vscale*  
maximum allowed vertical scaling factor

## Description

Calculate the vertical scaling factor as (*src* height) / (*dst* height).

If the calculated scaling factor is below *min\_vscale*, decrease the height of rectangle *dst* to compensate.

If the calculated scaling factor is above *max\_vscale*, decrease the height of rectangle *src* to compensate.

## RETURNS

The vertical scaling factor.

---

## Name

drm\_rect\_debug\_print — print the rectangle information

## Synopsis

```
void fsfuncdrm_rect_debug_print (r,
                                fixed_point);

const struct drm_rect * r;
bool fixed_point;
```

## Arguments

- r*  
rectangle to print
- fixed\_point*  
rectangle is in 16.16 fixed point format

## Flip-work Helper Reference



Util to queue up work to run from work-queue context after flip/vblank. Typically this can be used to defer unret of framebuffer's, cursor bo's, etc until after vblank. The APIs are all safe (and lockless) for up to one producer and once consumer at a time. The single-consumer aspect is ensured by committing the queued work to a single work-queue.

Name

struct drm\_flip\_work — flip work queue

Synopsis

```
struct drm_flip_work {
    const char * name;
    atomic_t pending;
    atomic_t count;
    drm_flip_func_t func;
    struct work_struct worker;
};
```

Members

name	
	debug name
pending	
	number of queued but not committed items
count	
	number of committed items
func	
	callback fxn called for each committed item
worker	
	worker which calls <i>func</i>

---

Name

drm\_flip\_work\_queue — queue work

Synopsis

```
void fsfuncdrm_flip_work_queue (work,
                                val);

struct drm_flip_work * work;
void * val;
```

Arguments

<i>work</i>	
	the flip-work
<i>val</i>	
	the value to queue

Description

Queues work, that will later be run (passed back to drm\_flip\_func\_t func) on a work queue after drm\_flip\_work\_commit is called.

---

Name

drm\_flip\_work\_commit — commit queued work

Synopsis

```
void fsfuncdrm_flip_work_commit (work,
                                 wq);

struct drm_flip_work * work;
struct workqueue_struct * wq;
```

## Arguments

*work*

the flip-work

*wq*

the work-queue to run the queued work on

## Description

Trigger work previously queued by `drm_flip_work_queue` to run on a workqueue. The typical usage would be to queue work (via `drm_flip_work_queue`) at any point (from vblank irq and/or prior), and then from vblank irq commit the queued work.

---

## Name

`drm_flip_work_init` — initialize flip-work

## Synopsis

```
int fsfuncdrm_flip_work_init (work,
                             size,
                             name,
                             func);

struct drm_flip_work * work;
int size;
const char * name;
drm_flip_func_t func;
```

## Arguments

*work*

the flip-work to initialize

*size*

the max queue depth

*name*

debug name

*func*

the callback work function

## Description

Initializes/allocates resources for the flip-work

## RETURNS

Zero on success, error code on failure.

---

## Name

`drm_flip_work_cleanup` — cleans up flip-work

## Synopsis

```
void fsfuncdrm_flip_work_cleanup (work);
struct drm_flip_work * work;
```

## Arguments

*work*

the flip-work to cleanup

## Description

Destroy resources allocated for the flip-work

## VMA Offset Manager

The vma-manager is responsible to map arbitrary driver-dependent memory regions into the linear user address-space. It provides offsets to the caller which can then be used on the `address_space` of the `drm-device`. It takes care to not overlap regions, size them appropriately and to not confuse mm-core by inconsistent fake `vm_pgoff` fields. Drivers shouldn't use this for object placement in VMEM. This manager should only be used to manage mappings into linear user-space VMs.

We use `drm_mm` as backend to manage object allocations. But it is highly optimized for `alloc/free` calls, not lookups. Hence, we use an rb-tree to speed up offset lookups.

You must not use multiple offset managers on a single `address_space`. Otherwise, mm-core will be unable to tear down memory mappings as the VM will no longer be linear. Please use `VM_NONLINEAR` in that case and implement your own offset managers.

This offset manager works on page-based addresses. That is, every argument and return code (with the exception of `drm_vma_node_offset_addr`) is given in number of pages, not number of bytes. That means, object sizes and offsets must always be page-aligned (as usual). If you want to get a valid byte-based user-space address for a given offset, please see `drm_vma_node_offset_addr`.

Additionally to offset management, the vma offset manager also handles access management. For every open-file context that is allowed to access a given node, you must call `drm_vma_node_allow`. Otherwise, an `mmap` call on this open-file with the offset of the node will fail with `-EACCES`. To revoke access again, use `drm_vma_node_revoke`. However, the caller is responsible for destroying already existing mappings, if required.

## Name

`drm_vma_offset_manager_init` — Initialize new offset-manager

## Synopsis

```
void fsfuncdrm_vma_offset_manager_init (mgr,
                                       page_offset,
                                       size);

struct drm_vma_offset_manager * mgr;
unsigned long page_offset;
unsigned long size;
```

## Arguments

*mgr*

Manager object

*page\_offset*

Offset of available memory area (page-based)

*size*

Size of available address space range (page-based)

## Description

Initialize a new offset-manager. The offset and area size available for the manager are given as *page\_offset* and *size*. Both are interpreted as page-numbers, not bytes.

Adding/removing nodes from the manager is locked internally and protected against concurrent access. However, node allocation and destruction is left for the caller. While calling into the vma-manager, a given node must always be guaranteed to be referenced.

---

## Name

`drm_vma_offset_manager_destroy` — Destroy offset manager

## Synopsis

```
void fsfuncdrm_vma_offset_manager_destroy (mgr);
struct drm_vma_offset_manager * mgr;
```

## Arguments

*mgr*

Manager object

## Description

Destroy an object manager which was previously created via `drm_vma_offset_manager_init`. The caller must remove all allocated nodes before destroying the manager. Otherwise, `drm_mm` will refuse to free the requested resources.

The manager must not be accessed after this function is called.

## Name

drm\_vma\_offset\_lookup — Find node in offset space

## Synopsis

```
struct drm_vma_offset_node * fsfuncdrm_vma_offset_lookup (mgr,  
                                                         start,  
                                                         pages);  
  
struct drm_vma_offset_manager * mgr;  
unsigned long start;  
unsigned long pages;
```

## Arguments

*mgr*

Manager object

*start*

Start address for object (page-based)

*pages*

Size of object (page-based)

## Description

Find a node given a start address and object size. This returns the `_best_` match for the given node. That is, *start* may point somewhere into a valid region and the given node will be returned, as long as the node spans the whole requested area (given the size in number of pages as *pages*).

## RETURNS

Returns NULL if no suitable node can be found. Otherwise, the best match is returned. It's the caller's responsibility to make sure the node doesn't get destroyed before the caller can access it.

---

## Name

drm\_vma\_offset\_lookup\_locked — Find node in offset space

## Synopsis

```
struct drm_vma_offset_node * fsfuncdrm_vma_offset_lookup_locked (mgr,  
                                                                start,  
                                                                pages);  
  
struct drm_vma_offset_manager * mgr;  
unsigned long start;  
unsigned long pages;
```

## Arguments

*mgr*

Manager object

*start*

Start address for object (page-based)

*pages*

Size of object (page-based)

## Description

Same as `drm_vma_offset_lookup` but requires the caller to lock offset lookup manually. See `drm_vma_offset_lock_lookup` for an example.

## RETURNS

Returns NULL if no suitable node can be found. Otherwise, the best match is returned.

---

## Name

## Synopsis

```
int fsfuncdrm_vma_offset_add (mgr,
                             node,
                             pages);
struct drm_vma_offset_manager * mgr;
struct drm_vma_offset_node * node;
unsigned long pages;
```

## Arguments

*mgr*

Manager object

*node*

Node to be added

*pages*

Allocation size visible to user-space (in number of pages)

## Description

Add a node to the offset-manager. If the node was already added, this does nothing and return 0. *pages* is the size of the object given in number of pages. After this call succeeds, you can access the offset of the node until it is removed again.

If this call fails, it is safe to retry the operation or call `drm_vma_offset_remove`, anyway. However, no cleanup is required in that case.

*pages* is not required to be the same size as the underlying memory object that you want to map. It only limits the size that user-space can map into their address space.

## RETURNS

0 on success, negative error code on failure.

---

## Name

drm\_vma\_offset\_remove — Remove offset node from manager

## Synopsis

```
void fsfuncdrm_vma_offset_remove (mgr,
                                  node);
struct drm_vma_offset_manager * mgr;
struct drm_vma_offset_node * node;
```

## Arguments

*mgr*

Manager object

*node*

Node to be removed

## Description

Remove a node from the offset manager. If the node wasn't added before, this does nothing. After this call returns, the offset and size will be 0 until a new offset is allocated via `drm_vma_offset_add` again. Helper functions like `drm_vma_node_start` and `drm_vma_node_offset_addr` will return 0 if no offset is allocated.

---

## Name

drm\_vma\_node\_allow — Add open-file to list of allowed users

## Synopsis

```
int fsfuncdrm_vma_node_allow (node,
                              filp);
struct drm_vma_offset_node * node;
```

```
struct file * filp;
```

## Arguments

*node*

Node to modify

*filp*

Open file to add

## Description

Add *filp* to the list of allowed open-files for this node. If *filp* is already on this list, the ref-count is incremented.

The list of allowed-users is preserved across `drm_vma_offset_add` and `drm_vma_offset_remove` calls. You may even call it if the node is currently not added to any offset-manager.

You must remove all open-files the same number of times as you added them before destroying the node. Otherwise, you will leak memory.

This is locked against concurrent access internally.

## RETURNS

0 on success, negative error code on internal failure (out-of-mem)

---

## Name

`drm_vma_node_revoke` — Remove open-file from list of allowed users

## Synopsis

```
void fsfuncdrm_vma_node_revoke (node,  
                                filp);  
struct drm_vma_offset_node * node;  
struct file * filp;
```

## Arguments

*node*

Node to modify

*filp*

Open file to remove

## Description

Decrement the ref-count of *filp* in the list of allowed open-files on *node*. If the ref-count drops to zero, remove *filp* from the list. You must call this once for every `drm_vma_node_allow` on *filp*.

This is locked against concurrent access internally.

If *filp* is not on the list, nothing is done.

---

## Name

`drm_vma_node_is_allowed` — Check whether an open-file is granted access

## Synopsis

```
bool fsfuncdrm_vma_node_is_allowed (node,  
                                    filp);  
struct drm_vma_offset_node * node;  
struct file * filp;
```

## Arguments

*node*

Node to check

*filp*

Open-file to check for

## Description

Search the list in *node* whether *filp* is currently on the list of allowed open-files (see `drm_vma_node_allow`).

This is locked against concurrent access internally.

## RETURNS

true iff *filp* is on the list

---

## Name

`drm_vma_offset_exact_lookup` — Look up node by exact address

## Synopsis

```
struct drm_vma_offset_node * fsfuncdrm_vma_offset_exact_lookup (mgr,  
                                                             start,  
                                                             pages);  
  
struct drm_vma_offset_manager * mgr;  
unsigned long start;  
unsigned long pages;
```

## Arguments

*mgr*

Manager object

*start*

Start address (page-based, not byte-based)

*pages*

Size of object (page-based)

## Description

Same as `drm_vma_offset_lookup` but does not allow any offset into the node. It only returns the exact object with the given start address.

## RETURNS

Node at exact start address *start*.

---

## Name

`drm_vma_offset_lock_lookup` — Lock lookup for extended private use

## Synopsis

```
void fsfuncdrm_vma_offset_lock_lookup (mgr);  
struct drm_vma_offset_manager * mgr;
```

## Arguments

*mgr*

Manager object

## Description

Lock VMA manager for extended lookups. Only `*_locked` VMA function calls are allowed while holding this lock. All other contexts are blocked from VMA until the lock is released via `drm_vma_offset_unlock_lookup`.

Use this if you need to take a reference to the objects returned by `drm_vma_offset_lookup_locked` before releasing this lock again.

This lock must not be used for anything else than extended lookups. You must not call any other VMA helpers while holding this lock.

## Note

You're in atomic-context while holding this lock!

## Example

```
drm_vma_offset_lock_lookup(mgr);
node = drm_vma_offset_lookup_locked(mgr);
if (node)
    kref_get_unless_zero(container_of(node, sth, entr));
drm_vma_offset_unlock_lookup(mgr);
```

---

## Name

drm\_vma\_offset\_unlock\_lookup — Unlock lookup for extended private use

## Synopsis

```
void fsfuncdrm_vma_offset_unlock_lookup (mgr);
struct drm_vma_offset_manager * mgr;
```

## Arguments

*mgr*

Manager object

## Description

Release lookup-lock. See `drm_vma_offset_lock_lookup` for more information.

---

## Name

drm\_vma\_node\_reset — Initialize or reset node object

## Synopsis

```
void fsfuncdrm_vma_node_reset (node);
struct drm_vma_offset_node * node;
```

## Arguments

*node*

Node to initialize or reset

## Description

Reset a node to its initial state. This must be called before using it with any VMA offset manager.

This must not be called on an already allocated node, or you will leak memory.

---

## Name

drm\_vma\_node\_start — Return start address for page-based addressing

## Synopsis

```
unsigned long fsfuncdrm_vma_node_start (node);
struct drm_vma_offset_node * node;
```

## Arguments

*node*

Node to inspect

## Description

Return the start address of the given node. This can be used as offset into the linear VM space that is provided by the VMA offset manager. Note that this can only be used for page-based addressing. If you need a proper offset for user-space mappings, you must apply “<< PAGE\_SHIFT” or use the `drm_vma_node_offset_addr` helper instead.



## RETURNS

Start address of *node* for page-based addressing. 0 if the node does not have an offset allocated.

---

### Name

drm\_vma\_node\_size — Return size (page-based)

### Synopsis

```
unsigned long fsfuncdrm_vma_node_size (node);
struct drm_vma_offset_node * node;
```

### Arguments

*node*

Node to inspect

### Description

Return the size as number of pages for the given node. This is the same size that was passed to `drm_vma_offset_add`. If no offset is allocated for the node, this is 0.

## RETURNS

Size of *node* as number of pages. 0 if the node does not have an offset allocated.

---

### Name

drm\_vma\_node\_has\_offset — Check whether node is added to offset manager

### Synopsis

```
bool fsfuncdrm_vma_node_has_offset (node);
struct drm_vma_offset_node * node;
```

### Arguments

*node*

Node to be checked

## RETURNS

true iff the node was previously allocated an offset and added to an vma offset manager.

---

### Name

drm\_vma\_node\_offset\_addr — Return sanitized offset for user-space mmaps

### Synopsis

```
__u64 fsfuncdrm_vma_node_offset_addr (node);
struct drm_vma_offset_node * node;
```

### Arguments

*node*

Linked offset node

### Description

Same as `drm_vma_node_start` but returns the address as a valid offset that can be used for user-space mappings during `mmap`. This must not be called on unlinked nodes.

## RETURNS

Offset of *node* for byte-based addressing. 0 if the node does not have an object allocated.

---

## Name

drm\_vma\_node\_unmap — Unmap offset node

## Synopsis

```
void fsfuncdrm_vma_node_unmap (node,  
                               file_mapping);  
struct drm_vma_offset_node * node;  
struct address_space * file_mapping;
```

## Arguments

*node*

Offset node

*file\_mapping*

Address space to unmap *node* from

## Description

Unmap all userspace mappings for a given offset node. The mappings must be associated with the *file\_mapping* address-space. If no offset exists or the address-space is invalid, nothing is done.

This call is unlocked. The caller must guarantee that `drm_vma_offset_remove` is not called on this node concurrently.

---

## Name

drm\_vma\_node\_verify\_access — Access verification helper for TTM

## Synopsis

```
int fsfuncdrm_vma_node_verify_access (node,  
                                     filp);  
struct drm_vma_offset_node * node;  
struct file * filp;
```

## Arguments

*node*

Offset node

*filp*

Open-file

## Description

This checks whether *filp* is granted access to *node*. It is the same as `drm_vma_node_is_allowed` but suitable as drop-in helper for TTM `verify_access` callbacks.

## RETURNS

0 if access is granted, -EACCES otherwise.

## KMS Properties

Drivers may need to expose additional parameters to applications than those described in the previous sections. KMS supports attaching properties to CRTC, connectors and planes and offers a userspace API to list, get and set the property values.

Properties are identified by a name that uniquely defines the property purpose, and store an associated value. For all property types except blob properties the value is a 64-bit unsigned integer.

KMS differentiates between properties and property instances. Drivers first create properties and then create and associate individual instances of those properties to objects. A property can be instantiated multiple times and associated with different objects. Values are stored in property instances, and all other property information are stored in the property and shared between all instances of the property.

Every property is created with a type that influences how the KMS core handles the property. Supported property types are

DRM\_MODE\_PROP\_RANGE

Range properties report their minimum and maximum admissible values. The KMS core verifies that values set by application fit in that range.

## DRM\_MODE\_PROP\_ENUM

Enumerated properties take a numerical value that ranges from 0 to the number of enumerated values defined by the property minus one, and associate a free-formed string name to each value. Applications can retrieve the list of defined value-name pairs and use the numerical value to get and set property instance values.

## DRM\_MODE\_PROP\_BITMASK

Bitmask properties are enumeration properties that additionally restrict all enumerated values to the 0..63 range. Bitmask property instance values combine one or more of the enumerated bits defined by the property.

## DRM\_MODE\_PROP\_BLOB

Blob properties store a binary blob without any format restriction. The binary blobs are created as KMS standalone objects, and blob property instance values store the ID of their associated blob object.

Blob properties are only used for the connector EDID property and cannot be created by drivers.

To create a property drivers call one of the following functions depending on the property type. All property creation functions take property flags and name, as well as type-specific arguments.

- `struct drm_property *drm_property_create_range(struct drm_device *dev, int flags, const char *name, uint64_t min, uint64_t max);`

Create a range property with the given minimum and maximum values.

- `struct drm_property *drm_property_create_enum(struct drm_device *dev, int flags, const char *name, const struct drm_prop_enum_list *props, int num_values);`

Create an enumerated property. The *props* argument points to an array of *num\_values* value-name pairs.

- `struct drm_property *drm_property_create_bitmask(struct drm_device *dev, int flags, const char *name, const struct drm_prop_enum_list *props, int num_values);`

Create a bitmask property. The *props* argument points to an array of *num\_values* value-name pairs.

Properties can additionally be created as immutable, in which case they will be read-only for applications but can be modified by the driver. To create an immutable property drivers must set the `DRM_MODE_PROP_IMMUTABLE` flag at property creation time.

When no array of value-name pairs is readily available at property creation time for enumerated or range properties, drivers can create the property using the `drm_property_create` function and manually add enumeration value-name pairs by calling the `drm_property_add_enum` function. Care must be taken to properly specify the property type through the *flags* argument.

After creating properties drivers can attach property instances to CRTC, connector and plane objects by calling the `drm_object_attach_property`. The function takes a pointer to the target object, a pointer to the previously created property and an initial instance value.

## Vertical Blanking

Vertical blanking plays a major role in graphics rendering. To achieve tear-free display, users must synchronize page flips and/or rendering 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 and 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 the `drm_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 load operation. The function will set the struct `drm_device` *vblank\_disable\_allowed* field to 0. This will keep vertical blanking interrupts enabled permanently until the first mode set operation, where *vblank\_disable\_allowed* is set to 1. The reason behind this is not clear. 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.

# Open/Close, File Operations and IOCTLs

## Open and Close

```
int (*firstopen) (struct drm_device *);
void (*lastclose) (struct drm_device *);
int (*open) (struct drm_device *, struct drm_file *);
void (*preclose) (struct drm_device *, struct drm_file *);
void (*postclose) (struct drm_device *, struct drm_file *);
```

Open and close handlers. None of those methods are mandatory.

The `firstopen` method is called by the DRM core for legacy UMS (User Mode Setting) drivers only when an application opens a device that has no other opened file handle. UMS drivers can implement it to acquire device resources. KMS drivers can't use the method and must acquire resources in the `load` method instead.

Similarly the `lastclose` method is called when the last application holding a file handle opened on the device closes it, for both UMS and KMS drivers. Additionally, the method is also called at module unload time or, for hot-pluggable devices, when the device is unplugged. The `firstopen` and `lastclose` calls can thus be unbalanced.

The `open` method is called every time the device is opened by an application. Drivers can allocate per-file private data in this method and store them in the struct `drm_file` `driver_priv` field. Note that the `open` method is called before `firstopen`.

The close operation is split into `preclose` and `postclose` methods. Drivers must stop and cleanup all per-file operations in the `preclose` method. For instance pending vertical blanking and page flip events must be cancelled. No per-file operation is allowed on the file handle after returning from the `preclose` method.

Finally the `postclose` method is called as the last step of the close operation, right before calling the `lastclose` method if no other open file handle exists for the device. Drivers that have allocated per-file private data in the `open` method should free it here.

The `lastclose` method should restore CRTC and plane properties to default value, so that a subsequent open of the device will not inherit state from the previous user. It can also be used to execute delayed power switching state changes, e.g. in conjunction with the `vga-switcheroo` infrastructure. Beyond that KMS drivers should not do any further cleanup. Only legacy UMS drivers might need to clean up device state so that the `vga` console or an independent `fbdev` driver could take over.

## File Operations

```
const struct file_operations *fops
```

File operations for the DRM device node.

Drivers must define the file operations structure that forms the DRM userspace API entry point, even though most of those operations are implemented in the DRM core. The `open`, `release` and `ioctl` operations are handled by

```
    .owner = THIS_MODULE,
    .open = drm_open,
    .release = drm_release,
    .unlocked_ioctl = drm_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = drm_compat_ioctl,
#endif
```

Drivers that implement private `ioctls` that requires 32/64bit compatibility support must provide their own `compat_ioctl` handler that processes private `ioctls` and calls `drm_compat_ioctl` for core `ioctls`.

The `read` and `poll` operations provide support for reading DRM events and polling them. They are implemented by

```
    .poll = drm_poll,
    .read = drm_read,
    .llseek = no_llseek,
```

The memory mapping implementation varies depending on how the driver manages memory. Pre-GEM drivers will use `drm_mmap`, while GEM-aware drivers will use `drm_gem_mmap`. See [the section called "The Graphics Execution Manager \(GEM\)"](#).

```
    .mmap = drm_gem_mmap,
```

No other file operation is supported by the DRM API.

## IOCTLs

```
struct drm_ioctl_desc *ioctls;
int num_ioctls;
```

Driver-specific `ioctls` descriptors table.

Driver-specific `ioctls` numbers start at `DRM_COMMAND_BASE`. The `ioctls` descriptors table is indexed by the `ioctl` number offset from the base value. Drivers can use the `DRM_IOCTL_DEF_DRV()` macro to initialize the table entries.

```
DRM_IOCTL_DEF_DRV(ioctl, func, flags)
```

`ioctl` is the `ioctl` name. Drivers must define the `DRM_##ioctl` and `DRM_IOCTL_##ioctl` macros to the `ioctl` number offset from `DRM_COMMAND_BASE` and the `ioctl` number respectively. The first macro is private to the device while the second must be exposed to userspace in a public header.

`func` is a pointer to the `ioctl` handler function compatible with the `drm_ioctl_t` type.

```
typedef int drm_ioctl_t(struct drm_device *dev, void *data,
    struct drm_file *file_priv);
```

*flags* is a bitmask combination of the following values. It restricts how the ioctl is allowed to be called.

- DRM\_AUTH - Only authenticated callers allowed
- DRM\_MASTER - The ioctl can only be called on the master file handle
- DRM\_ROOT\_ONLY - Only callers with the SYSADMIN capability allowed
- DRM\_CONTROL\_ALLOW - The ioctl can only be called on a control device
- DRM\_UNLOCKED - The ioctl handler will be called without locking the DRM global mutex

## Command submission & fencing

This should cover a few device-specific command submission implementations.

## Suspend/Resume

The DRM core provides some suspend/resume code, but drivers wanting full suspend/resume support should provide `save()` and `restore()` functions. These are called at suspend, hibernate, or resume time, and should perform any state save or restore required by your device across suspend or hibernate states.

```
int (*suspend) (struct drm_device *, pm_message_t state);
int (*resume) (struct drm_device *);
```

Those are legacy suspend and resume methods. New driver should use the power management interface provided by their bus type (usually through the struct `device_driver dev_pm_ops`) and set these methods to NULL.

## DMA services

This should cover how DMA mapping etc. is supported by the core. These functions are deprecated and should not be used.

## Chapter 3. Userland interfaces

### Table of Contents

[Render nodes](#)

[VBlank event handling](#)

The DRM core exports several interfaces to applications, generally intended to be used through corresponding libdrm wrapper functions. In addition, drivers export device-specific interfaces for use by userspace drivers & device-aware applications through ioctls and sysfs files.

External interfaces include: memory mapping, context management, DMA operations, AGP management, vblank control, fence management, memory management, and output management.

Cover generic ioctls and sysfs layout here. We only need high-level info, since man pages should cover the rest.

## 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>`. Additionally, 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. However, the planned KMS control interface has never been written and so the control node stays unused to date.

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. But the DRM API required unprivileged clients to authenticate to a DRM-Master prior to getting GPU access. To avoid this step and to grant clients GPU access without authenticating, render nodes were introduced. Render nodes solely serve render clients, that is, 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. Additionally to this basic interface, drivers must mark their driver-dependent render-only ioctls as `<term>DRM_RENDER_ALLOW</term>` so render clients can use them. Driver authors must be careful 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. Instead, 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.

## 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).

# Appendix A. DRM Driver API

Include auto-generated API reference here (need to reference it from paragraphs above too).