

Linux DRM Developer's Guide

Jesse Barnes, Intel Corporation <jesse.barnes@intel.com>

Laurent Pinchart, Ideas on board SPRL

<laurent.pinchart@ideasonboard.com>

Daniel Vetter, Intel Corporation <daniel.vetter@ffwll.ch>

Linux DRM Developer's Guide

by Jesse Barnes, Laurent Pinchart, and Daniel Vetter
Copyright © 2008-2009, 2013-2014 Intel Corporation
Copyright © 2012 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.

Table of Contents

I. DRM Core	1
1. Introduction	4
2. DRM Internals	5
Driver Initialization	5
Driver Information	5
Device Registration	6
Driver Load	21
Memory management	23
The Translation Table Manager (TTM)	23
The Graphics Execution Manager (GEM)	24
VMA Offset Manager	41
PRIME Buffer Sharing	60
PRIME Function References	61
DRM MM Range Allocator	69
DRM MM Range Allocator Function References	70
Mode Setting	92
Display Modes Function Reference	92
Frame Buffer Creation	115
Dumb Buffer Objects	116
Output Polling	117
Locking	117
KMS Initialization and Cleanup	117
CRTCs (struct drm_crtc)	117
Planes (struct drm_plane)	119
Encoders (struct drm_encoder)	120
Connectors (struct drm_connector)	121
Cleanup	124
Output discovery and initialization example	124
KMS API Functions	125
KMS Locking	180
Mode Setting Helper Functions	191
Helper Functions	191
CRTC Helper Operations	192
Encoder Helper Operations	193
Connector Helper Operations	193
Modeset Helper Functions Reference	196
Output Probing Helper Functions Reference	205
fbdev Helper Functions Reference	213
Display Port Helper Functions Reference	228
EDID Helper Functions Reference	242
Rectangle Utilities Reference	263
Flip-work Helper Reference	278
HDMI Infoframes Helper Reference	283
Plane Helper Reference	293
KMS Properties	299
Existing KMS Properties	300
Vertical Blanking	308
Vertical Blanking and Interrupt Handling Functions Reference	308
Open/Close, File Operations and IOCTLs	330
Open and Close	330
File Operations	331

IOCTLs	331
Legacy Support Code	332
Legacy Suspend/Resume	332
Legacy DMA Services	332
3. Userland interfaces	333
Render nodes	333
VBlank event handling	333
II. DRM Drivers	335
4. drm/i915 Intel GFX Driver	337
Display Hardware Handling	337
Mode Setting Infrastructure	337
Plane Configuration	337
Output Probing	337
DPIO	337
Memory Management and Command Submission	338
Batchbuffer Parsing	338

List of Tables

2.1.	300
4.1. Dual channel PHY (VLV/CHV)	338
4.2. Single channel PHY (CHV)	338

Part I. DRM Core

This first part of the DRM Developer's Guide documents core DRM code, helper libraries for writing drivers and generic userspace interfaces exposed by DRM drivers.

Table of Contents

1. Introduction	4
2. DRM Internals	5
Driver Initialization	5
Driver Information	5
Device Registration	6
Driver Load	21
Memory management	23
The Translation Table Manager (TTM)	23
The Graphics Execution Manager (GEM)	24
VMA Offset Manager	41
PRIME Buffer Sharing	60
PRIME Function References	61
DRM MM Range Allocator	69
DRM MM Range Allocator Function References	70
Mode Setting	92
Display Modes Function Reference	92
Frame Buffer Creation	115
Dumb Buffer Objects	116
Output Polling	117
Locking	117
KMS Initialization and Cleanup	117
CRTCs (struct drm_crtc)	117
Planes (struct drm_plane)	119
Encoders (struct drm_encoder)	120
Connectors (struct drm_connector)	121
Cleanup	124
Output discovery and initialization example	124
KMS API Functions	125
KMS Locking	180
Mode Setting Helper Functions	191
Helper Functions	191
CRTC Helper Operations	192
Encoder Helper Operations	193
Connector Helper Operations	193
Modeset Helper Functions Reference	196
Output Probing Helper Functions Reference	205
fbdev Helper Functions Reference	213
Display Port Helper Functions Reference	228
EDID Helper Functions Reference	242
Rectangle Utilities Reference	263
Flip-work Helper Reference	278
HDMI Infoframes Helper Reference	283
Plane Helper Reference	293
KMS Properties	299
Existing KMS Properties	300
Vertical Blanking	308
Vertical Blanking and Interrupt Handling Functions Reference	308
Open/Close, File Operations and IOCTLs	330
Open and Close	330
File Operations	331
IOCTLs	331

Legacy Support Code	332
Legacy Suspend/Resume	332
Legacy DMA Services	332
3. Userland interfaces	333
Render nodes	333
VBlank event handling	333

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

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.

Newer drivers that no longer require a `drm_bus` structure can alternatively use the low-level device initialization and registration functions such as `drm_dev_alloc()` and `drm_dev_register()` directly.

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

<code>DRIVER_USE_AGP</code>	Driver uses AGP interface, the DRM core will manage AGP resources.
<code>DRIVER_REQUIRE_AGP</code>	Driver needs AGP interface to function. AGP initialization failure will become a fatal error.
<code>DRIVER_PCI_DMA</code>	Driver is capable of PCI DMA, mapping of PCI DMA buffers to userspace will be enabled. Deprecated.
<code>DRIVER_SG</code>	Driver can perform scatter/gather DMA, allocation and mapping of scatter/gather buffers will be enabled. Deprecated.
<code>DRIVER_HAVE_DMA</code>	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.

Device Registration

A number of functions are provided to help with device registration. The functions deal with PCI, USB and platform devices, respectively.

Name

`drm_pci_alloc` — Allocate a PCI consistent memory block, for DMA.

Synopsis

```
drm_dma_handle_t * drm_pci_alloc (struct drm_device * dev, size_t size,  
size_t align);
```

Arguments

dev DRM device

size size of block to allocate

align alignment of block

Return

A handle to the allocated memory block on success or NULL on failure.

Name

`drm_pci_free` — Free a PCI consistent memory block

Synopsis

```
void drm_pci_free (struct drm_device * dev, drm_dma_handle_t * dma);
```

Arguments

dev DRM device

dma handle to memory block

Name

`drm_get_pci_dev` — Register a PCI device with the DRM subsystem

Synopsis

```
int drm_get_pci_dev (struct pci_dev * pdev, const struct pci_device_id  
* ent, struct drm_driver * driver);
```

Arguments

pdev PCI device

ent entry from the PCI ID table that matches *pdev*

driver DRM device driver

Description

Attempt to get inter module “drm” information. If we are first then register the character device and inter module information. Try and register, if we fail to register, backout previous work.

Return

0 on success or a negative error code on failure.

Name

`drm_pci_init` — Register matching PCI devices with the DRM subsystem

Synopsis

```
int drm_pci_init (struct drm_driver * driver, struct pci_driver *  
pdriver);
```

Arguments

driver DRM device driver

pdriver PCI device driver

Description

Initializes a `drm_device` structures, registering the stubs and initializing the AGP device.

Return

0 on success or a negative error code on failure.

Name

`drm_pci_exit` — Unregister matching PCI devices from the DRM subsystem

Synopsis

```
void drm_pci_exit (struct drm_driver * driver, struct pci_driver *  
pdriver);
```

Arguments

driver DRM device driver

pdriver PCI device driver

Description

Unregisters one or more devices matched by a PCI driver from the DRM subsystem.

Name

`drm_usb_init` — Register matching USB devices with the DRM subsystem

Synopsis

```
int drm_usb_init (struct drm_driver * driver, struct usb_driver *  
udriver);
```

Arguments

driver DRM device driver

udriver USB device driver

Description

Registers one or more devices matched by a USB driver with the DRM subsystem.

Return

0 on success or a negative error code on failure.

Name

`drm_usb_exit` — Unregister matching USB devices from the DRM subsystem

Synopsis

```
void drm_usb_exit (struct drm_driver * driver, struct usb_driver *  
udriver);
```

Arguments

driver DRM device driver

udriver USB device driver

Description

Unregisters one or more devices matched by a USB driver from the DRM subsystem.

Name

`drm_platform_init` — Register a platform device with the DRM subsystem

Synopsis

```
int    drm_platform_init    (struct    drm_driver    *    driver,    struct
platform_device * platform_device);
```

Arguments

driver DRM device driver

platform_device platform device to register

Description

Registers the specified DRM device driver and platform device with the DRM subsystem, initializing a `drm_device` structure and calling the driver's `.load` function.

Return

0 on success or a negative error code on failure.

New drivers that no longer rely on the services provided by the `drm_bus` structure can call the low-level device registration functions directly. The `drm_dev_alloc()` function can be used to allocate and initialize a new `drm_device` structure. Drivers will typically want to perform some additional setup on this structure, such as allocating driver-specific data and storing a pointer to it in the DRM device's `dev_private` field. Drivers should also set the device's unique name using the `drm_dev_set_unique()` function. After it has been set up a device can be registered with the DRM subsystem by calling `drm_dev_register()`. This will cause the device to be exposed to userspace and will call the driver's `.load()` implementation. When a device is removed, the DRM device can safely be unregistered and freed by calling `drm_dev_unregister()` followed by a call to `drm_dev_unref()`.

Name

`drm_put_dev` — Unregister and release a DRM device

Synopsis

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

Arguments

dev DRM device

Description

Called at module unload time or when a PCI device is unplugged.

Use of this function is discouraged. It will eventually go away completely. Please use `drm_dev_unregister` and `drm_dev_unref` explicitly instead.

Cleans up all DRM device, calling `drm_lastclose`.

Name

`drm_dev_alloc` — Allocate new DRM device

Synopsis

```
struct drm_device * drm_dev_alloc (struct drm_driver * driver, struct  
device * parent);
```

Arguments

driver DRM driver to allocate device for

parent Parent device object

Description

Allocate and initialize a new DRM device. No device registration is done. Call `drm_dev_register` to advertise the device to user space and register it with other core subsystems.

The initial ref-count of the object is 1. Use `drm_dev_ref` and `drm_dev_unref` to take and drop further ref-counts.

RETURNS

Pointer to new DRM device, or NULL if out of memory.

Name

`drm_dev_ref` — Take reference of a DRM device

Synopsis

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

Arguments

dev device to take reference of or NULL

Description

This increases the ref-count of *dev* by one. You *must* already own a reference when calling this. Use `drm_dev_unref` to drop this reference again.

This function never fails. However, this function does not provide *any* guarantee whether the device is alive or running. It only provides a reference to the object and the memory associated with it.

Name

`drm_dev_unref` — Drop reference of a DRM device

Synopsis

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

Arguments

dev device to drop reference of or NULL

Description

This decreases the ref-count of *dev* by one. The device is destroyed if the ref-count drops to zero.

Name

`drm_dev_register` — Register DRM device

Synopsis

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

Arguments

dev Device to register

flags Flags passed to the driver's `.load` function

Description

Register the DRM device *dev* with the system, advertise device to user-space and start normal device operation. *dev* must be allocated via `drm_dev_alloc` previously.

Never call this twice on any device!

RETURNS

0 on success, negative error code on failure.

Name

`drm_dev_unregister` — Unregister DRM device

Synopsis

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

Arguments

dev Device to unregister

Description

Unregister the DRM device from the system. This does the reverse of `drm_dev_register` but does not deallocate the device. The caller must call `drm_dev_unref` to drop their final reference.

Name

`drm_dev_set_unique` — Set the unique name of a DRM device

Synopsis

```
int drm_dev_set_unique (struct drm_device * dev, const char * fmt, ...);
```

Arguments

dev device of which to set the unique name

fmt format string for unique name

... variable arguments

Description

Sets the unique name of a DRM device using the specified format string and a variable list of arguments. Drivers can use this at driver probe time if the unique name of the devices they drive is static.

Return

0 on success or a negative error code on failure.

Driver Load

The `load` method is the driver and device initialization entry point. The method is responsible for allocating and initializing driver private data, 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 Data

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.

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

`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 passed-in 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* [<http://lwn.net/Articles/283798/>] 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.

GEM also supports buffer sharing with dma-buf file descriptors through PRIME. GEM-based drivers must use the provided helpers functions to implement the exporting and importing correctly. See the section called "PRIME Buffer Sharing". Since sharing file descriptors is inherently more secure than the easily guessable and global GEM names it is the preferred buffer sharing mechanism. Sharing buffers through GEM names is only supported for legacy userspace. Furthermore PRIME also allows cross-device buffer sharing since it is based on dma-bufs.

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.

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.

GEM Function Reference

Name

`drm_gem_object_init` — initialize an allocated shmem-backed GEM object

Synopsis

```
int drm_gem_object_init (struct drm_device * dev, struct drm_gem_object  
* obj, size_t size);
```

Arguments

dev drm_device the object should be initialized for

obj drm_gem_object to initialize

size object size

Description

Initialize an already allocated GEM object of the specified size with shmfs backing store.

Name

`drm_gem_private_object_init` — initialize an allocated private GEM object

Synopsis

```
void drm_gem_private_object_init (struct drm_device * dev, struct
drm_gem_object * obj, size_t size);
```

Arguments

dev drm_device the object should be initialized for

obj drm_gem_object to initialize

size object size

Description

Initialize an already allocated GEM object of the specified size with no GEM provided backing store. Instead the caller is responsible for backing the object and handling it.

Name

`drm_gem_handle_delete` — deletes the given file-private handle

Synopsis

```
int drm_gem_handle_delete (struct drm_file * filp, u32 handle);
```

Arguments

filp drm file-private structure to use for the handle look up

handle userspace handle to delete

Description

Removes the GEM handle from the *filp* lookup table and if this is the last handle also cleans up linked resources like GEM names.

Name

`drm_gem_dumb_destroy` — dumb fb callback helper for gem based drivers

Synopsis

```
int drm_gem_dumb_destroy (struct drm_file * file, struct drm_device *  
dev, uint32_t handle);
```

Arguments

file drm file-private structure to remove the dumb handle from

dev corresponding `drm_device`

handle the dumb handle to remove

Description

This implements the `->dumb_destroy` kms driver callback for drivers which use gem to manage their backing storage.

Name

`drm_gem_handle_create` — create a gem handle for an object

Synopsis

```
int drm_gem_handle_create (struct drm_file * file_priv, struct
drm_gem_object * obj, u32 * handlep);
```

Arguments

file_priv drm file-private structure to register the handle for

obj object to register

handlep pionter to return the created handle to the caller

Description

Create a handle for this object. This adds a handle reference to the object, which includes a regular reference count. Callers will likely want to dereference the object afterwards.

Name

`drm_gem_free_mmap_offset` — release a fake mmap offset for an object

Synopsis

```
void drm_gem_free_mmap_offset (struct drm_gem_object * obj);
```

Arguments

obj obj in question

Description

This routine frees fake offsets allocated by `drm_gem_create_mmap_offset`.

Name

`drm_gem_create_mmap_offset_size` — create a fake mmap offset for an object

Synopsis

```
int drm_gem_create_mmap_offset_size (struct drm_gem_object * obj, size_t
size);
```

Arguments

obj obj in question

size the virtual size

Description

GEM memory mapping works by handing back to userspace a fake mmap offset it can use in a subsequent `mmap(2)` call. The DRM core code then looks up the object based on the offset and sets up the various memory mapping structures.

This routine allocates and attaches a fake offset for *obj*, in cases where the virtual size differs from the physical size (ie. `obj->size`). Otherwise just use `drm_gem_create_mmap_offset`.

Name

`drm_gem_create_mmap_offset` — create a fake mmap offset for an object

Synopsis

```
int drm_gem_create_mmap_offset (struct drm_gem_object * obj);
```

Arguments

obj obj in question

Description

GEM memory mapping works by handing back to userspace a fake mmap offset it can use in a subsequent `mmap(2)` call. The DRM core code then looks up the object based on the offset and sets up the various memory mapping structures.

This routine allocates and attaches a fake offset for *obj*.

Name

`drm_gem_get_pages` — helper to allocate backing pages for a GEM object from shmem

Synopsis

```
struct page ** drm_gem_get_pages (struct drm_gem_object * obj, gfp_t  
gfpmask);
```

Arguments

obj obj in question

gfpmask gfp mask of requested pages

Name

`drm_gem_put_pages` — helper to free backing pages for a GEM object

Synopsis

```
void drm_gem_put_pages (struct drm_gem_object * obj, struct page **  
pages, bool dirty, bool accessed);
```

Arguments

<i>obj</i>	obj in question
<i>pages</i>	pages to free
<i>dirty</i>	if true, pages will be marked as dirty
<i>accessed</i>	if true, the pages will be marked as accessed

Name

`drm_gem_object_free` — free a GEM object

Synopsis

```
void drm_gem_object_free (struct kref * kref);
```

Arguments

kref kref of the object to free

Description

Called after the last reference to the object has been lost. Must be called holding `struct_mutex`

Frees the object

Name

`drm_gem_mmap_obj` — memory map a GEM object

Synopsis

```
int drm_gem_mmap_obj (struct drm_gem_object * obj, unsigned long
obj_size, struct vm_area_struct * vma);
```

Arguments

obj the GEM object to map

obj_size the object size to be mapped, in bytes

vma VMA for the area to be mapped

Description

Set up the VMA to prepare mapping of the GEM object using the `gem_vm_ops` provided by the driver. Depending on their requirements, drivers can either provide a fault handler in their `gem_vm_ops` (in which case any accesses to the object will be trapped, to perform migration, GTT binding, surface register allocation, or performance monitoring), or mmap the buffer memory synchronously after calling `drm_gem_mmap_obj`.

This function is mainly intended to implement the DMABUF mmap operation, when the GEM object is not looked up based on its fake offset. To implement the DRM mmap operation, drivers should use the `drm_gem_mmap` function.

`drm_gem_mmap_obj` assumes the user is granted access to the buffer while `drm_gem_mmap` prevents unprivileged users from mapping random objects. So callers must verify access restrictions before calling this helper.

NOTE

This function has to be protected with `dev->struct_mutex`

Return 0 or success or `-EINVAL` if the object size is smaller than the VMA size, or if no `gem_vm_ops` are provided.

Name

`drm_gem_mmap` — memory map routine for GEM objects

Synopsis

```
int drm_gem_mmap (struct file * filp, struct vm_area_struct * vma);
```

Arguments

filp DRM file pointer

vma VMA for the area to be mapped

Description

If a driver supports GEM object mapping, `mmap` calls on the DRM file descriptor will end up here.

Look up the GEM object based on the offset passed in (`vma->vm_pgoff` will contain the fake offset we created when the GTT map ioctl was called on the object) and map it with a call to `drm_gem_mmap_obj`.

If the caller is not granted access to the buffer object, the `mmap` will fail with `EACCES`. Please see the vma manager for more information.

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 drm_vma_offset_manager_init (struct drm_vma_offset_manager * mgr,  
unsigned long page_offset, unsigned long size);
```

Arguments

<i>mgr</i>	Manager object
<i>page_offset</i>	Offset of available memory area (page-based)
<i>size</i>	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 drm_vma_offset_manager_destroy (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    *    drm_vma_offset_lookup    (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  *  drm_vma_offset_lookup_locked (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

`drm_vma_offset_add` — Add offset node to manager

Synopsis

```
int drm_vma_offset_add (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 drm_vma_offset_remove (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 drm_vma_node_allow (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 drm_vma_node_revoke (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 drm_vma_node_is_allowed (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  * drm_vma_offset_exact_lookup (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 drm_vma_offset_lock_lookup (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 drm_vma_offset_unlock_lookup (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 drm_vma_node_reset (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 drm_vma_node_start (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 drm_vma_node_size (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 drm_vma_node_has_offset (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 drm_vma_node_offset_addr (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 drm_vma_node_unmap (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 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 drm_vma_node_verify_access (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.

PRIME Buffer Sharing

PRIME is the cross device buffer sharing framework in drm, originally created for the OPTIMUS range of multi-gpu platforms. To userspace PRIME buffers are dma-buf based file descriptors.

Overview and Driver Interface

Similar to GEM global names, PRIME file descriptors are also used to share buffer 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 the 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. Similar to the mode setting API PRIME is agnostic to the underlying buffer object manager, as long as handles are 32bit unsigned integers.

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 PRIME.

PRIME Helper Functions

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

PRIME Function References

Name

`drm_gem_dmabuf_release` — `dma_buf` release implementation for GEM

Synopsis

```
void drm_gem_dmabuf_release (struct dma_buf * dma_buf);
```

Arguments

dma_buf buffer to be released

Description

Generic release function for `dma_bufs` exported as PRIME buffers. GEM drivers must use this in their `dma_buf` ops structure as the release callback.

Name

`drm_gem_prime_export` — helper library implementation of the export callback

Synopsis

```
struct dma_buf * drm_gem_prime_export (struct drm_device * dev, struct
drm_gem_object * obj, int flags);
```

Arguments

dev `drm_device` to export from

obj GEM object to export

flags flags like `DRM_CLOEXEC`

Description

This is the implementation of the `gem_prime_export` functions for GEM drivers using the PRIME helpers.

Name

`drm_gem_prime_handle_to_fd` — PRIME export function for GEM drivers

Synopsis

```
int drm_gem_prime_handle_to_fd (struct drm_device * dev, struct drm_file  
* file_priv, uint32_t handle, uint32_t flags, int * prime_fd);
```

Arguments

<i>dev</i>	dev to export the buffer from
<i>file_priv</i>	drm file-private structure
<i>handle</i>	buffer handle to export
<i>flags</i>	flags like <code>DRM_CLOEXEC</code>
<i>prime_fd</i>	pointer to storage for the fd id of the create dma-buf

Description

This is the PRIME export function which must be used mandatorily by GEM drivers to ensure correct lifetime management of the underlying GEM object. The actual exporting from GEM object to a dma-buf is done through the `gem_prime_export` driver callback.

Name

`drm_gem_prime_import` — helper library implementation of the import callback

Synopsis

```
struct drm_gem_object * drm_gem_prime_import (struct drm_device * dev,  
struct dma_buf * dma_buf);
```

Arguments

dev `drm_device` to import into

dma_buf `dma-buf` object to import

Description

This is the implementation of the `gem_prime_import` functions for GEM drivers using the PRIME helpers.

Name

`drm_gem_prime_fd_to_handle` — PRIME import function for GEM drivers

Synopsis

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

Arguments

<i>dev</i>	dev to export the buffer from
<i>file_priv</i>	drm file-private structure
<i>prime_fd</i>	fd id of the dma-buf which should be imported
<i>handle</i>	pointer to storage for the handle of the imported buffer object

Description

This is the PRIME import function which must be used mandatorily by GEM drivers to ensure correct lifetime management of the underlying GEM object. The actual importing of GEM object from the dma-buf is done through the `gem_import_export` driver callback.

Name

`drm_prime_pages_to_sg` — converts a page array into an sg list

Synopsis

```
struct sg_table * drm_prime_pages_to_sg (struct page ** pages, int
nr_pages);
```

Arguments

pages pointer to the array of page pointers to convert

nr_pages length of the page vector

Description

This helper creates an sg table object from a set of pages the driver is responsible for mapping the pages into the importers address space for use with `dma_buf` itself.

Name

`drm_prime_sg_to_page_addr_arrays` — convert an sg table into a page array

Synopsis

```
int drm_prime_sg_to_page_addr_arrays (struct sg_table * sgt, struct page
** pages, dma_addr_t * addrs, int max_pages);
```

Arguments

<i>sgt</i>	scatter-gather table to convert
<i>pages</i>	array of page pointers to store the page array in
<i>addrs</i>	optional array to store the dma bus address of each page
<i>max_pages</i>	size of both the passed-in arrays

Description

Exports an sg table into an array of pages and addresses. This is currently required by the TTM driver in order to do correct fault handling.

Name

`drm_prime_gem_destroy` — helper to clean up a PRIME-imported GEM object

Synopsis

```
void drm_prime_gem_destroy (struct drm_gem_object * obj, struct sg_table  
* sg);
```

Arguments

obj GEM object which was created from a dma-buf

sg the sg-table which was pinned at import time

Description

This is the cleanup functions which GEM drivers need to call when they use `drm_gem_prime_import` to import dma-bufs.

DRM MM Range Allocator

Overview

`drm_mm` provides a simple range allocator. The drivers are free to use the resource allocator from the linux core if it suits them, the upside of `drm_mm` is that it's in the DRM core. Which means that it's easier to extend for some of the crazier special purpose needs of gpus.

The main data struct is `drm_mm`, allocations are tracked in `drm_mm_node`. Drivers are free to embed either of them into their own suitable datastructures. `drm_mm` itself will not do any allocations of its own, so if drivers choose not to embed nodes they need to still allocate them themselves.

The range allocator also supports reservation of preallocated blocks. This is useful for taking over initial mode setting configurations from the firmware, where an object needs to be created which exactly matches the firmware's scanout target. As long as the range is still free it can be inserted anytime after the allocator is initialized, which helps with avoiding looped dependencies in the driver load sequence.

`drm_mm` maintains a stack of most recently freed holes, which of all simplistic datastructures seems to be a fairly decent approach to clustering allocations and avoiding too much fragmentation. This means free space searches are $O(\text{num_holes})$. Given that all the fancy features `drm_mm` supports something better would be fairly complex and since gfx thrashing is a fairly steep cliff not a real concern. Removing a node again is $O(1)$.

`drm_mm` supports a few features: Alignment and range restrictions can be supplied. Further more every `drm_mm_node` has a color value (which is just an opaque unsigned long) which in conjunction with a driver callback can be used to implement sophisticated placement restrictions. The i915 DRM driver uses this to implement guard pages between incompatible caching domains in the graphics TT.

Two behaviors are supported for searching and allocating: bottom-up and top-down. The default is bottom-up. Top-down allocation can be used if the memory area has different restrictions, or just to reduce fragmentation.

Finally iteration helpers to walk all nodes and all holes are provided as are some basic allocator dumpers for debugging.

LRU Scan/Eviction Support

Very often GPUs need to have continuous allocations for a given object. When evicting objects to make space for a new one it is therefore not most efficient when we simply start to select all objects from the tail of an LRU until there's a suitable hole: Especially for big objects or nodes that otherwise have special allocation constraints there's a good chance we evict lots of (smaller) objects unnecessarily.

The DRM range allocator supports this use-case through the scanning interfaces. First a scan operation needs to be initialized with `drm_mm_init_scan` or `drm_mm_init_scan_with_range`. The driver adds objects to the roaster (probably by walking an LRU list, but this can be freely implemented) until a suitable hole is found or there's no further evitable object.

The driver must walk through all objects again in exactly the reverse order to restore the allocator state. Note that while the allocator is used in the scan mode no other operation is allowed.

Finally the driver evicts all objects selected in the scan. Adding and removing an object is $O(1)$, and since freeing a node is also $O(1)$ the overall complexity is $O(\text{scanned_objects})$. So like the free stack which needs to be walked before a scan operation even begins this is linear in the number of objects. It doesn't seem to hurt badly.

DRM MM Range Allocator Function References

Name

`drm_mm_reserve_node` — insert an pre-initialized node

Synopsis

```
int drm_mm_reserve_node (struct drm_mm * mm, struct drm_mm_node * node);
```

Arguments

mm `drm_mm` allocator to insert *node* into

node `drm_mm_node` to insert

Description

This functions inserts an already set-up `drm_mm_node` into the allocator, meaning that start, size and color must be set by the caller. This is useful to initialize the allocator with preallocated objects which must be set-up before the range allocator can be set-up, e.g. when taking over a firmware framebuffer.

Returns

0 on success, -ENOSPC if there's no hole where *node* is.

Name

`drm_mm_insert_node_generic` — search for space and insert *node*

Synopsis

```
int drm_mm_insert_node_generic (struct drm_mm * mm, struct drm_mm_node  
* node, unsigned long size, unsigned alignment, unsigned long color,  
enum drm_mm_search_flags sflags, enum drm_mm_allocator_flags aflags);
```

Arguments

<i>mm</i>	drm_mm to allocate from
<i>node</i>	preallocate node to insert
<i>size</i>	size of the allocation
<i>alignment</i>	alignment of the allocation
<i>color</i>	opaque tag value to use for this node
<i>sflags</i>	flags to fine-tune the allocation search
<i>aflags</i>	flags to fine-tune the allocation behavior

Description

The preallocated node must be cleared to 0.

Returns

0 on success, -ENOSPC if there's no suitable hole.

Name

`drm_mm_insert_node_in_range_generic` — ranged search for space and insert *node*

Synopsis

```
int drm_mm_insert_node_in_range_generic (struct drm_mm * mm, struct
drm_mm_node * node, unsigned long size, unsigned alignment, unsigned long
color, unsigned long start, unsigned long end, enum drm_mm_search_flags
sflags, enum drm_mm_allocator_flags aflags);
```

Arguments

<i>mm</i>	drm_mm to allocate from
<i>node</i>	preallocate node to insert
<i>size</i>	size of the allocation
<i>alignment</i>	alignment of the allocation
<i>color</i>	opaque tag value to use for this node
<i>start</i>	start of the allowed range for this node
<i>end</i>	end of the allowed range for this node
<i>sflags</i>	flags to fine-tune the allocation search
<i>aflags</i>	flags to fine-tune the allocation behavior

Description

The preallocated node must be cleared to 0.

Returns

0 on success, -ENOSPC if there's no suitable hole.

Name

`drm_mm_remove_node` — Remove a memory node from the allocator.

Synopsis

```
void drm_mm_remove_node (struct drm_mm_node * node);
```

Arguments

node `drm_mm_node` to remove

Description

This just removes a node from its `drm_mm` allocator. The node does not need to be cleared again before it can be re-inserted into this or any other `drm_mm` allocator. It is a bug to call this function on a un-allocated node.

Name

`drm_mm_replace_node` — move an allocation from *old* to *new*

Synopsis

```
void drm_mm_replace_node (struct drm_mm_node * old, struct drm_mm_node  
* new);
```

Arguments

old `drm_mm_node` to remove from the allocator

new `drm_mm_node` which should inherit *old*'s allocation

Description

This is useful for when drivers embed the `drm_mm_node` structure and hence can't move allocations by reassigning pointers. It's a combination of remove and insert with the guarantee that the allocation start will match.

Name

`drm_mm_init_scan` — initialize lru scanning

Synopsis

```
void drm_mm_init_scan (struct drm_mm * mm, unsigned long size, unsigned  
alignment, unsigned long color);
```

Arguments

<i>mm</i>	drm_mm to scan
<i>size</i>	size of the allocation
<i>alignment</i>	alignment of the allocation
<i>color</i>	opaque tag value to use for the allocation

Description

This simply sets up the scanning routines with the parameters for the desired hole. Note that there's no need to specify allocation flags, since they only change the place a node is allocated from within a suitable hole.

Warning

As long as the scan list is non-empty, no other operations than adding/removing nodes to/from the scan list are allowed.

Name

`drm_mm_init_scan_with_range` — initialize range-restricted lru scanning

Synopsis

```
void drm_mm_init_scan_with_range (struct drm_mm * mm, unsigned long
size, unsigned alignment, unsigned long color, unsigned long start,
unsigned long end);
```

Arguments

<i>mm</i>	drm_mm to scan
<i>size</i>	size of the allocation
<i>alignment</i>	alignment of the allocation
<i>color</i>	opaque tag value to use for the allocation
<i>start</i>	start of the allowed range for the allocation
<i>end</i>	end of the allowed range for the allocation

Description

This simply sets up the scanning routines with the parameters for the desired hole. Note that there's no need to specify allocation flags, since they only change the place a node is allocated from within a suitable hole.

Warning

As long as the scan list is non-empty, no other operations than adding/removing nodes to/from the scan list are allowed.

Name

`drm_mm_scan_add_block` — add a node to the scan list

Synopsis

```
bool drm_mm_scan_add_block (struct drm_mm_node * node);
```

Arguments

node `drm_mm_node` to add

Description

Add a node to the scan list that might be freed to make space for the desired hole.

Returns

True if a hole has been found, false otherwise.

Name

`drm_mm_scan_remove_block` — remove a node from the scan list

Synopsis

```
bool drm_mm_scan_remove_block (struct drm_mm_node * node);
```

Arguments

node `drm_mm_node` to remove

Description

Nodes `_must_` be removed in the exact same order from the scan list as they have been added, otherwise the internal state of the memory manager will be corrupted.

When the scan list is empty, the selected memory nodes can be freed. An immediately following `drm_mm_search_free` with `!DRM_MM_SEARCH_BEST` will then return the just freed block (because its at the top of the `free_stack` list).

Returns

True if this block should be evicted, false otherwise. Will always return false when no hole has been found.

Name

`drm_mm_clean` — checks whether an allocator is clean

Synopsis

```
bool drm_mm_clean (struct drm_mm * mm);
```

Arguments

mm `drm_mm` allocator to check

Returns

True if the allocator is completely free, false if there's still a node allocated in it.

Name

`drm_mm_init` — initialize a drm-mm allocator

Synopsis

```
void drm_mm_init (struct drm_mm * mm, unsigned long start, unsigned  
long size);
```

Arguments

mm the `drm_mm` structure to initialize

start start of the range managed by *mm*

size end of the range managed by *mm*

Description

Note that *mm* must be cleared to 0 before calling this function.

Name

`drm_mm_takedown` — clean up a `drm_mm` allocator

Synopsis

```
void drm_mm_takedown (struct drm_mm * mm);
```

Arguments

mm `drm_mm` allocator to clean up

Description

Note that it is a bug to call this function on an allocator which is not clean.

Name

`drm_mm_debug_table` — dump allocator state to dmesg

Synopsis

```
void drm_mm_debug_table (struct drm_mm * mm, const char * prefix);
```

Arguments

mm drm_mm allocator to dump

prefix prefix to use for dumping to dmesg

Name

`drm_mm_dump_table` — dump allocator state to a `seq_file`

Synopsis

```
int drm_mm_dump_table (struct seq_file * m, struct drm_mm * mm);
```

Arguments

m `seq_file` to dump to

mm `drm_mm` allocator to dump

Name

`drm_mm_node_allocated` — checks whether a node is allocated

Synopsis

```
bool drm_mm_node_allocated (struct drm_mm_node * node);
```

Arguments

node `drm_mm_node` to check

Description

Drivers should use this helpers for proper encapsulation of `drm_mm` internals.

Returns

True if the *node* is allocated.

Name

`drm_mm_initialized` — checks whether an allocator is initialized

Synopsis

```
bool drm_mm_initialized (struct drm_mm * mm);
```

Arguments

mm `drm_mm` to check

Description

Drivers should use this helpers for proper encapsulation of `drm_mm` internals.

Returns

True if the *mm* is initialized.

Name

`drm_mm_hole_node_start` — computes the start of the hole following *node*

Synopsis

```
unsigned long drm_mm_hole_node_start (struct drm_mm_node * hole_node);
```

Arguments

hole_node `drm_mm_node` which implicitly tracks the following hole

Description

This is useful for driver-specific debug dumpers. Otherwise drivers should not inspect holes themselves. Drivers must check first whether a hole indeed follows by looking at `node->hole_follows`.

Returns

Start of the subsequent hole.

Name

`drm_mm_hole_node_end` — computes the end of the hole following *node*

Synopsis

```
unsigned long drm_mm_hole_node_end (struct drm_mm_node * hole_node);
```

Arguments

hole_node `drm_mm_node` which implicitly tracks the following hole

Description

This is useful for driver-specific debug dumpers. Otherwise drivers should not inspect holes themselves. Drivers must check first whether a hole indeed follows by looking at `node->hole_follows`.

Returns

End of the subsequent hole.

Name

`drm_mm_for_each_node` — iterator to walk over all allocated nodes

Synopsis

```
drm_mm_for_each_node ( entry, mm );
```

Arguments

entry `drm_mm_node` structure to assign to in each iteration step

mm `drm_mm` allocator to walk

Description

This iterator walks over all nodes in the range allocator. It is implemented with `list_for_each`, so not safe against removal of elements.

Name

`drm_mm_for_each_hole` — iterator to walk over all holes

Synopsis

```
drm_mm_for_each_hole ( entry, mm, hole_start, hole_end );
```

Arguments

<i>entry</i>	<code>drm_mm_node</code> used internally to track progress
<i>mm</i>	<code>drm_mm</code> allocator to walk
<i>hole_start</i>	ulong variable to assign the hole start to on each iteration
<i>hole_end</i>	ulong variable to assign the hole end to on each iteration

Description

This iterator walks over all holes in the range allocator. It is implemented with `list_for_each`, so not save against removal of elements. *entry* is used internally and will not reflect a real `drm_mm_node` for the very first hole. Hence users of this iterator may not access it.

Implementation Note

We need to inline `list_for_each_entry` in order to be able to set `hole_start` and `hole_end` on each iteration while keeping the macro sane.

The `__drm_mm_for_each_hole` version is similar, but with added support for going backwards.

Name

`drm_mm_insert_node` — search for space and insert *node*

Synopsis

```
int drm_mm_insert_node (struct drm_mm * mm, struct drm_mm_node * node,
unsigned long size, unsigned alignment, enum drm_mm_search_flags flags);
```

Arguments

<i>mm</i>	drm_mm to allocate from
<i>node</i>	preallocate node to insert
<i>size</i>	size of the allocation
<i>alignment</i>	alignment of the allocation
<i>flags</i>	flags to fine-tune the allocation

Description

This is a simplified version of `drm_mm_insert_node_generic` with *color* set to 0.

The preallocated node must be cleared to 0.

Returns

0 on success, -ENOSPC if there's no suitable hole.

Name

`drm_mm_insert_node_in_range` — ranged search for space and insert *node*

Synopsis

```
int drm_mm_insert_node_in_range (struct drm_mm * mm, struct drm_mm_node
* node, unsigned long size, unsigned alignment, unsigned long start,
unsigned long end, enum drm_mm_search_flags flags);
```

Arguments

<i>mm</i>	drm_mm to allocate from
<i>node</i>	preallocate node to insert
<i>size</i>	size of the allocation
<i>alignment</i>	alignment of the allocation
<i>start</i>	start of the allowed range for this node
<i>end</i>	end of the allowed range for this node
<i>flags</i>	flags to fine-tune the allocation

Description

This is a simplified version of `drm_mm_insert_node_in_range_generic` with *color* set to 0.

The preallocated node must be cleared to 0.

Returns

0 on success, -ENOSPC if there's no suitable hole.

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.

Display Modes Function Reference

Name

`drm_mode_is_stereo` — check for stereo mode flags

Synopsis

```
bool drm_mode_is_stereo (const struct drm_display_mode * mode);
```

Arguments

mode `drm_display_mode` to check

Returns

True if the mode is one of the stereo modes (like side-by-side), false if not.

Name

`drm_mode_debug_printmodeline` — print a mode to dmesg

Synopsis

```
void drm_mode_debug_printmodeline (const struct drm_display_mode *  
mode);
```

Arguments

mode mode to print

Description

Describe *mode* using `DRM_DEBUG`.

Name

`drm_mode_create` — create a new display mode

Synopsis

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

Arguments

dev DRM device

Description

Create a new, cleared `drm_display_mode` with `kzalloc`, allocate an ID for it and return it.

Returns

Pointer to new mode on success, `NULL` on error.

Name

`drm_mode_destroy` — remove a mode

Synopsis

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

Arguments

dev DRM device

mode mode to remove

Description

Release *mode*'s unique ID, then free it *mode* structure itself using `kfree`.

Name

`drm_mode_probed_add` — add a mode to a connector's `probed_mode` list

Synopsis

```
void drm_mode_probed_add (struct drm_connector * connector, struct
drm_display_mode * mode);
```

Arguments

connector connector the new mode

mode mode data

Description

Add *mode* to *connector*'s `probed_mode` list for later use. This list should then in a second step get filtered and all the modes actually supported by the hardware moved to the *connector*'s modes list.

Name

`drm_cvt_mode` — create a modeline based on the CVT algorithm

Synopsis

```
struct drm_display_mode * drm_cvt_mode (struct drm_device * dev, int
hdisplay, int vdisplay, int vrefresh, bool reduced, bool interlaced,
bool margins);
```

Arguments

<i>dev</i>	drm device
<i>hdisplay</i>	hdisplay size
<i>vdisplay</i>	vdisplay size
<i>vrefresh</i>	vrefresh rate
<i>reduced</i>	whether to use reduced blanking
<i>interlaced</i>	whether to compute an interlaced mode
<i>margins</i>	whether to add margins (borders)

Description

This function is called to generate the modeline based on CVT algorithm according to the `hdisplay`, `vdisplay`, `vrefresh`. It is based from the VESA(TM) Coordinated Video Timing Generator by Graham Loveridge April 9, 2003 available at

http

[//www.elo.utfsm.cl/~elo212/docs/CVTd6r1.xls](http://www.elo.utfsm.cl/~elo212/docs/CVTd6r1.xls)

And it is copied from `xf86CVTmode` in `xserver/hw/xfree86/modes/xf86cvt.c`. What I have done is to translate it by using integer calculation.

Returns

The modeline based on the CVT algorithm stored in a `drm_display_mode` object. The display mode object is allocated with `drm_mode_create`. Returns `NULL` when no mode could be allocated.

Name

`drm_gtf_mode_complex` — create the modeline based on the full GTF algorithm

Synopsis

```
struct drm_display_mode * drm_gtf_mode_complex (struct drm_device * dev,  
int hdisplay, int vdisplay, int vrefresh, bool interlaced, int margins,  
int GTF_M, int GTF_2C, int GTF_K, int GTF_2J);
```

Arguments

<i>dev</i>	drm device
<i>hdisplay</i>	hdisplay size
<i>vdisplay</i>	vdisplay size
<i>vrefresh</i>	vrefresh rate.
<i>interlaced</i>	whether to compute an interlaced mode
<i>margins</i>	desired margin (borders) size
<i>GTF_M</i>	extended GTF formula parameters
<i>GTF_2C</i>	extended GTF formula parameters
<i>GTF_K</i>	extended GTF formula parameters
<i>GTF_2J</i>	extended GTF formula parameters

Description

GTF feature blocks specify C and J in multiples of 0.5, so we pass them in here multiplied by two. For a C of 40, pass in 80.

Returns

The modeline based on the full GTF algorithm stored in a `drm_display_mode` object. The display mode object is allocated with `drm_mode_create`. Returns NULL when no mode could be allocated.

Name

`drm_gtf_mode` — create the modeline based on the GTF algorithm

Synopsis

```
struct drm_display_mode * drm_gtf_mode (struct drm_device * dev, int
hdisplay, int vdisplay, int vrefresh, bool interlaced, int margins);
```

Arguments

<i>dev</i>	drm device
<i>hdisplay</i>	hdisplay size
<i>vdisplay</i>	vdisplay size
<i>vrefresh</i>	vrefresh rate.
<i>interlaced</i>	whether to compute an interlaced mode
<i>margins</i>	desired margin (borders) size

Description

return the modeline based on GTF algorithm

This function is to create the modeline based on the GTF algorithm.

Generalized Timing Formula is derived from

GTF Spreadsheet by Andy Morrish (1/5/97)

available at [http](http://www.vesa.org)

[//www.vesa.org](http://www.vesa.org)

And it is copied from the file of `xserver/hw/xfree86/modes/xf86gtf.c`. What I have done is to translate it by using integer calculation. I also refer to the function of `fb_get_mode` in the file of `drivers/video/fbmon.c`

Standard GTF parameters

$M = 600$ $C = 40$ $K = 128$ $J = 20$

Returns

The modeline based on the GTF algorithm stored in a `drm_display_mode` object. The display mode object is allocated with `drm_mode_create`. Returns `NULL` when no mode could be allocated.

Name

`drm_display_mode_from_videomode` — fill in *dmode* using *vm*,

Synopsis

```
void drm_display_mode_from_videomode (const struct videomode * vm,  
struct drm_display_mode * dmode);
```

Arguments

vm videomode structure to use as source

dmode `drm_display_mode` structure to use as destination

Description

Fills out *dmode* using the display mode specified in *vm*.

Name

`of_get_drm_display_mode` — get a `drm_display_mode` from devicetree

Synopsis

```
int of_get_drm_display_mode (struct device_node * np, struct
drm_display_mode * dmode, int index);
```

Arguments

np device_node with the timing specification

dmode will be set to the return value

index index into the list of display timings in devicetree

Description

This function is expensive and should only be used, if only one mode is to be read from DT. To get multiple modes start with `of_get_display_timings` and work with that instead.

Returns

0 on success, a negative errno code when no of videomode node was found.

Name

`drm_mode_set_name` — set the name on a mode

Synopsis

```
void drm_mode_set_name (struct drm_display_mode * mode);
```

Arguments

mode name will be set in this mode

Description

Set the name of *mode* to a standard format which is <hdisplay>x<vdisplay> with an optional 'i' suffix for interlaced modes.

Name

`drm_mode_vrefresh` — get the vrefresh of a mode

Synopsis

```
int drm_mode_vrefresh (const struct drm_display_mode * mode);
```

Arguments

mode mode

Returns

mode's vrefresh rate in Hz, rounded to the nearest integer. Calculates the value first if it is not yet set.

Name

`drm_mode_set_crtcinfo` — set CRTC modesetting timing parameters

Synopsis

```
void    drm_mode_set_crtcinfo    (struct    drm_display_mode    *    p,    int  
    adjust_flags);
```

Arguments

p mode

adjust_flags a combination of adjustment flags

Description

Setup the CRTC modesetting timing parameters for *p*, adjusting if necessary.

- The `CRTC_INTERLACE_HALVE_V` flag can be used to halve vertical timings of interlaced modes. -
The `CRTC_STEREO_DOUBLE` flag can be used to compute the timings for buffers containing two eyes
(only adjust the timings when needed, eg. for “frame packing” or “side by side full”).

Name

`drm_mode_copy` — copy the mode

Synopsis

```
void drm_mode_copy (struct drm_display_mode * dst, const struct
drm_display_mode * src);
```

Arguments

dst mode to overwrite

src mode to copy

Description

Copy an existing mode into another mode, preserving the object id and list head of the destination mode.

Name

`drm_mode_duplicate` — allocate and duplicate an existing mode

Synopsis

```
struct drm_display_mode * drm_mode_duplicate (struct drm_device * dev,  
const struct drm_display_mode * mode);
```

Arguments

dev `drm_device` to allocate the duplicated mode for

mode `mode` to duplicate

Description

Just allocate a new mode, copy the existing mode into it, and return a pointer to it. Used to create new instances of established modes.

Returns

Pointer to duplicated mode on success, NULL on error.

Name

`drm_mode_equal` — test modes for equality

Synopsis

```
bool drm_mode_equal (const struct drm_display_mode * mode1, const struct
drm_display_mode * mode2);
```

Arguments

mode1 first mode

mode2 second mode

Description

Check to see if *mode1* and *mode2* are equivalent.

Returns

True if the modes are equal, false otherwise.

Name

`drm_mode_equal_no_clocks_no_stereo` — test modes for equality

Synopsis

```
bool drm_mode_equal_no_clocks_no_stereo (const struct drm_display_mode
* mode1, const struct drm_display_mode * mode2);
```

Arguments

mode1 first mode

mode2 second mode

Description

Check to see if *mode1* and *mode2* are equivalent, but don't check the pixel clocks nor the stereo layout.

Returns

True if the modes are equal, false otherwise.

Name

`drm_mode_validate_size` — make sure modes adhere to size constraints

Synopsis

```
void drm_mode_validate_size (struct drm_device * dev, struct list_head  
* mode_list, int maxX, int maxY);
```

Arguments

<i>dev</i>	DRM device
<i>mode_list</i>	list of modes to check
<i>maxX</i>	maximum width
<i>maxY</i>	maximum height

Description

This function is a helper which can be used to validate modes against size limitations of the DRM device/connector. If a mode is too big its status member is updated with the appropriate validation failure code. The list itself is not changed.

Name

`drm_mode_prune_invalid` — remove invalid modes from mode list

Synopsis

```
void drm_mode_prune_invalid (struct drm_device * dev, struct list_head  
* mode_list, bool verbose);
```

Arguments

<i>dev</i>	DRM device
<i>mode_list</i>	list of modes to check
<i>verbose</i>	be verbose about it

Description

This helper function can be used to prune a display mode list after validation has been completed. All modes whose status is not `MODE_OK` will be removed from the list, and if *verbose* the status code and mode name is also printed to `dmesg`.

Name

`drm_mode_sort` — sort mode list

Synopsis

```
void drm_mode_sort (struct list_head * mode_list);
```

Arguments

mode_list list of `drm_display_mode` structures to sort

Description

Sort *mode_list* by favorability, moving good modes to the head of the list.

Name

`drm_mode_connector_list_update` — update the mode list for the connector

Synopsis

```
void drm_mode_connector_list_update (struct drm_connector * connector,  
bool merge_type_bits);
```

Arguments

connector the connector to update

merge_type_bits whether to merge or overright type bits.

Description

This moves the modes from the *connector* `probed_modes` list to the actual mode list. It compares the probed mode against the current list and only adds different/new modes.

This is just a helper functions doesn't validate any modes itself and also doesn't prune any invalid modes. Callers need to do that themselves.

Name

`drm_mode_parse_command_line_for_connector` — parse command line modeline for connector

Synopsis

```
bool    drm_mode_parse_command_line_for_connector    (const    char    *  
mode_option, struct drm_connector * connector, struct drm_cmdline_mode  
* mode);
```

Arguments

mode_option optional per connector mode option

connector connector to parse modeline for

mode preallocated `drm_cmdline_mode` structure to fill out

Description

This parses *mode_option* command line modeline for modes and options to configure the connector. If *mode_option* is NULL the default command line modeline in `fb_mode_option` will be parsed instead.

This uses the same parameters as the fb modedb.c, except for an extra force-enable, force-enable-digital and force-disable bit at the end:

```
<xres>x<yres>[M][R][-<bpp>][@<refresh>][i][m][eDd]
```

The intermediate `drm_cmdline_mode` structure is required to store additional options from the command line modline like the force-enabel/disable flag.

Returns

True if a valid modeline has been parsed, false otherwise.

Name

`drm_mode_create_from_cmdline_mode` — convert a command line modeline into a DRM display mode

Synopsis

```
struct drm_display_mode * drm_mode_create_from_cmdline_mode (struct
drm_device * dev, struct drm_cmdline_mode * cmd);
```

Arguments

dev DRM device to create the new mode for

cmd input command line modeline

Returns

Pointer to converted mode on success, NULL on error.

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. For drivers using GEM as their userspace buffer management interface this would be a GEM handle. Drivers are however free to use their own backing storage object handles, e.g. `vmwgfx` directly exposes special TTM handles to userspace and so expects TTM handles in the create ioctl and not GEM handles.

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`.

Dumb Buffer Objects

The KMS API doesn't standardize backing storage object creation and leaves it to driver-specific ioctls. Furthermore actually creating a buffer object even for GEM-based drivers is done through a driver-specific ioctl - GEM only has a common userspace interface for sharing and destroying objects. 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 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 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 driver object (GEM or TTM handle) 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 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 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 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”.

Note that dumb objects may not be used for gpu acceleration, as has been attempted on some ARM embedded platforms. Such drivers really must have a hardware-specific ioctl to allocate suitable buffer objects.

Output Polling

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

This operation notifies the driver that the status of one or more connectors has changed. Drivers that use the fb helper can just call the `drm_fb_helper_hotplug_event` function to handle this operation.

Locking

Beside some lookup structures with their own locking (which is hidden behind the interface functions) most of the modeset state is protected by the `dev-<mode_config.lock` mutex and additionally per-crtc locks to allow cursor updates, pageflips and similar operations to occur concurrently with background tasks like output detection. Operations which cross domains like a full modeset always grab all locks. Drivers there need to protect resources shared between crtcs with additional locking. They also need to be careful to always grab the relevant crtc locks if a modset functions touches crtc state, e.g. for load detection (which does only grab the `mode_config.lock` to allow concurrent screen updates on live crtcs).

KMS Initialization and Cleanup

A KMS device is abstracted and exposed as a set of planes, CRTCs, encoders 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

Note that the drm core has no notion of restoring the mode setting state after resume, since all resume handling is in the full responsibility of the driver. The common mode setting helper library though provides a helper which can be used for this: `drm_helper_resume_force_mode`.

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, ul6 *r, ul6 *g, ul6 *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.

The DRM core recognizes three types of planes:

- `DRM_PLANE_TYPE_PRIMARY` represents a "main" plane for a CRTC. Primary planes are the planes operated upon by by CRTC modesetting and flipping operations described in the section called “CRTC Operations”.
- `DRM_PLANE_TYPE_CURSOR` represents a "cursor" plane for a CRTC. Cursor planes are the planes operated upon by the `DRM_IOCTL_MODE_CURSOR` and `DRM_IOCTL_MODE_CURSOR2` ioctls.
- `DRM_PLANE_TYPE_OVERLAY` represents all non-primary, non-cursor planes. Some drivers refer to these types of planes as "sprites" internally.

For compatibility with legacy userspace, only overlay planes are made available to userspace by default. Userspace clients may set the `DRM_CLIENT_CAP_UNIVERSAL_PLANES` client capability bit to indicate that they wish to receive a universal plane list containing all plane types.

Plane Initialization

To create a plane, a KMS drivers allocates and zeroes an instances of `struct drm_plane` (possibly as part of a larger structure) and registers it with a call to `drm_universal_plane_init`. The function takes

a bitmask of the CRTC's that can be associated with the plane, a pointer to the plane functions, a list of format supported formats, and the type of plane (primary, cursor, or overlay) being initialized.

Cursor and overlay planes are optional. All drivers should provide one primary plane per CRTC (although this requirement may change in the future); drivers that do not wish to provide special handling for primary planes may make use of the helper functions described in the section called “Plane Helper Reference” to create and register a primary plane with standard capabilities.

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 CRTC's, 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 CRTC's 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.

<i>interlace_allowed</i>	Whether the connector can handle interlaced modes.
<i>doublescan_allowed</i>	Whether the connector can handle doublescan.
<i>display_info</i>	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 <i>display_info.width_mm</i> and <i>display_info.height_mm</i> fields with the physical size of the display.
<i>polled</i>	Connector polling mode, a combination of <code>DRM_CONNECTOR_POLL_HPD</code> The connector generates hotplug events and doesn't need to be periodically polled. The <code>CONNECT</code> 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 CRTCs (`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 cleanup 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 drm_modeset_lock_all (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. Locks must be dropped with `drm_modeset_unlock_all`.

Name

`drm_modeset_unlock_all` — drop all modeset locks

Synopsis

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

Arguments

dev device

Description

This function drop all modeset locks taken by `drm_modeset_lock_all`.

Name

`drm_warn_on_modeset_not_all_locked` — check that all modeset locks are locked

Synopsis

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

Arguments

dev device

Description

Useful as a debug assert.

Name

`drm_get_connector_status_name` — return a string for connector status

Synopsis

```
const char * drm_get_connector_status_name (enum drm_connector_status
status);
```

Arguments

status connector status to compute name of

Description

In contrast to the other `drm_get_*_name` functions this one here returns a const pointer and hence is threadsafe.

Name

`drm_get_subpixel_order_name` — return a string for a given subpixel enum

Synopsis

```
const char * drm_get_subpixel_order_name (enum subpixel_order order);
```

Arguments

order enum of subpixel_order

Description

Note you could abuse this and return something out of bounds, but that would be a caller error. No unscrubbed user data should make it here.

Name

`drm_get_format_name` — return a string for drm fourcc format

Synopsis

```
const char * drm_get_format_name (uint32_t format);
```

Arguments

format format to compute name of

Description

Note that the buffer used by this function is globally shared and owned by the function itself.

FIXME

This isn't really multithreading safe.

Name

`drm_mode_object_find` — look up a drm object with static lifetime

Synopsis

```
struct drm_mode_object * drm_mode_object_find (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. Even with `DRM_MODE_OBJECT_ANY` (although that will simply return `NULL` rather than `WARN_ON`).

Name

`drm_framebuffer_init` — initialize a framebuffer

Synopsis

```
int    drm_framebuffer_init    (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 * drm_framebuffer_lookup (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, using `drm_framebuffer_unreference`.

Name

`drm_framebuffer_unreference` — unref a framebuffer

Synopsis

```
void drm_framebuffer_unreference (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 drm_framebuffer_reference (struct drm_framebuffer * fb);
```

Arguments

fb framebuffer

Description

This functions increments the fb's refcount.

Name

`drm_framebuffer_unregister_private` — unregister a private fb from the lookup idr

Synopsis

```
void drm_framebuffer_unregister_private (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 drm_framebuffer_cleanup (struct drm_framebuffer * fb);
```

Arguments

fb framebuffer to remove

Description

Cleanup framebuffer. This function is intended to be used from the drivers `->destroy` callback. It can also be used to clean up driver private framebuffers embedded into a larger structure.

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 drm_framebuffer_remove (struct drm_framebuffer * fb);
```

Arguments

fb framebuffer to remove

Description

Scans all the CRTC's 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_with_planes` — Initialise a new CRTC object with specified primary and cursor planes.

Synopsis

```
int drm_crtc_init_with_planes (struct drm_device * dev, struct drm_crtc
* crtc, struct drm_plane * primary, void * cursor, const struct
drm_crtc_funcs * funcs);
```

Arguments

<i>dev</i>	DRM device
<i>crtc</i>	CRTC object to init
<i>primary</i>	Primary plane for CRTC
<i>cursor</i>	Cursor plane for CRTC
<i>funcs</i>	callbacks for the new CRTC

Description

Initialises 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 drm_crtc_cleanup (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_crtc_index` — find the index of a registered CRTC

Synopsis

```
unsigned int drm_crtc_index (struct drm_crtc * crtc);
```

Arguments

crtc CRTC to find index for

Description

Given a registered CRTC, return the index of that CRTC within a DRM device's list of CRTCs.

Name

`drm_connector_init` — Init a preallocated connector

Synopsis

```
int drm_connector_init (struct drm_device * dev, struct drm_connector
* connector, const struct drm_connector_funcs * funcs, int
connector_type);
```

Arguments

<i>dev</i>	DRM device
<i>connector</i>	the connector to init
<i>funcs</i>	callbacks for this connector
<i>connector_type</i>	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 drm_connector_cleanup (struct drm_connector * connector);
```

Arguments

connector connector to cleanup

Description

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

Name

`drm_connector_unplug_all` — unregister connector userspace interfaces

Synopsis

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

Arguments

dev drm device

Description

This function unregisters all connector userspace interfaces in sysfs. Should be call when the device is disconnected, e.g. from an usb driver's ->disconnect callback.

Name

`drm_bridge_init` — initialize a drm transcoder/bridge

Synopsis

```
int drm_bridge_init (struct drm_device * dev, struct drm_bridge * bridge,  
const struct drm_bridge_funcs * funcs);
```

Arguments

<i>dev</i>	drm device
<i>bridge</i>	transcoder/bridge to set up
<i>funcs</i>	bridge function table

Description

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

Returns

Zero on success, error code on failure.

Name

`drm_bridge_cleanup` — cleans up an initialised bridge

Synopsis

```
void drm_bridge_cleanup (struct drm_bridge * bridge);
```

Arguments

bridge bridge to cleanup

Description

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

Name

`drm_encoder_init` — Init a preallocated encoder

Synopsis

```
int drm_encoder_init (struct drm_device * dev, struct drm_encoder *  
encoder, const struct drm_encoder_funcs * funcs, int encoder_type);
```

Arguments

<i>dev</i>	drm device
<i>encoder</i>	the encoder to init
<i>funcs</i>	callbacks for this encoder
<i>encoder_type</i>	user visible type of the encoder

Description

Initialises a preallocated encoder. Encoder should be subclassed as part of driver encoder objects.

Returns

Zero on success, error code on failure.

Name

`drm_encoder_cleanup` — cleans up an initialised encoder

Synopsis

```
void drm_encoder_cleanup (struct drm_encoder * encoder);
```

Arguments

encoder encoder to cleanup

Description

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

Name

`drm_universal_plane_init` — Initialize a new universal plane object

Synopsis

```
int drm_universal_plane_init (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, enum
drm_plane_type type);
```

Arguments

<i>dev</i>	DRM device
<i>plane</i>	plane object to init
<i>possible_crtcs</i>	bitmask of possible CRTCs
<i>funcs</i>	callbacks for the new plane
<i>formats</i>	array of supported formats (<code>DRM_FORMAT_*</code>)
<i>format_count</i>	number of elements in <i>formats</i>
<i>type</i>	type of plane (overlay, primary, cursor)

Description

Initializes a plane object of type *type*.

Returns

Zero on success, error code on failure.

Name

`drm_plane_init` — Initialize a legacy plane

Synopsis

```
int drm_plane_init (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 is_primary);
```

Arguments

<i>dev</i>	DRM device
<i>plane</i>	plane object to init
<i>possible_crtcs</i>	bitmask of possible CRTCs
<i>funcs</i>	callbacks for the new plane
<i>formats</i>	array of supported formats (<code>DRM_FORMAT_*</code>)
<i>format_count</i>	number of elements in <i>formats</i>
<i>is_primary</i>	plane type (primary vs overlay)

Description

Legacy API to initialize a DRM plane.

New drivers should call `drm_universal_plane_init` instead.

Returns

Zero on success, error code on failure.

Name

`drm_plane_cleanup` — Clean up the core plane usage

Synopsis

```
void drm_plane_cleanup (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 drm_plane_force_disable (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_dvi_i_properties` — create DVI-I specific connector properties

Synopsis

```
int drm_mode_create_dvi_i_properties (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  drm_mode_create_tv_properties (struct  drm_device  *  dev,  int
num_modes,  char  *  modes[ ] );
```

Arguments

<i>dev</i>	DRM device
<i>num_modes</i>	number of different TV formats (modes) supported
<i>modes[]</i>	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 drm_mode_create_scaling_mode_property (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 drm_mode_create_dirty_info_property (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 drm_mode_set_config_internal (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.

Returns

Zero on success, `errno` on failure.

Name

`drm_crtc_check_viewport` — Checks that a framebuffer is big enough for the CRTC viewport

Synopsis

```
int drm_crtc_check_viewport (const struct drm_crtc * crtc, int x, int
                             y, const struct drm_display_mode * mode, const struct drm_framebuffer
                             * fb);
```

Arguments

crtc CRTC that framebuffer will be displayed on

x x panning

y y panning

mode mode that framebuffer will be displayed under

fb framebuffer to check size of

Name

`drm_mode_legacy_fb_format` — compute drm fourcc code from legacy description

Synopsis

```
uint32_t drm_mode_legacy_fb_format (uint32_t bpp, uint32_t depth);
```

Arguments

bpp bits per pixels

depth bit depth per pixel

Description

Computes a drm fourcc pixel format code for the given *bpp/depth* values. Useful in fbdev emulation code, since that deals in those values.

Name

`drm_property_create` — create a new property type

Synopsis

```
struct drm_property * drm_property_create (struct drm_device * dev, int
flags, const char * name, int num_values);
```

Arguments

<i>dev</i>	drm device
<i>flags</i>	flags specifying the property type
<i>name</i>	name of the property
<i>num_values</i>	number of pre-defined values

Description

This creates a new generic drm property which can then be attached to a drm object with `drm_object_attach_property`. The returned property object must be freed with `drm_property_destroy`.

Returns

A pointer to the newly created property on success, NULL on failure.

Name

`drm_property_create_enum` — create a new enumeration property type

Synopsis

```
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);
```

Arguments

<i>dev</i>	drm device
<i>flags</i>	flags specifying the property type
<i>name</i>	name of the property
<i>props</i>	enumeration lists with property values
<i>num_values</i>	number of pre-defined values

Description

This creates a new generic drm property which can then be attached to a drm object with `drm_object_attach_property`. The returned property object must be freed with `drm_property_destroy`.

Userspace is only allowed to set one of the predefined values for enumeration properties.

Returns

A pointer to the newly created property on success, NULL on failure.

Name

`drm_property_create_bitmask` — create a new bitmask property type

Synopsis

```
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);
```

Arguments

<i>dev</i>	drm device
<i>flags</i>	flags specifying the property type
<i>name</i>	name of the property
<i>props</i>	enumeration lists with property bitflags
<i>num_values</i>	number of pre-defined values

Description

This creates a new generic drm property which can then be attached to a drm object with `drm_object_attach_property`. The returned property object must be freed with `drm_property_destroy`.

Compared to plain enumeration properties userspace is allowed to set any or'ed together combination of the predefined property bitflag values

Returns

A pointer to the newly created property on success, NULL on failure.

Name

`drm_property_create_range` — create a new ranged property type

Synopsis

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

Arguments

<i>dev</i>	drm device
<i>flags</i>	flags specifying the property type
<i>name</i>	name of the property
<i>min</i>	minimum value of the property
<i>max</i>	maximum value of the property

Description

This creates a new generic drm property which can then be attached to a drm object with `drm_object_attach_property`. The returned property object must be freed with `drm_property_destroy`.

Userspace is allowed to set any interger value in the (min, max) range inclusive.

Returns

A pointer to the newly created property on success, NULL on failure.

Name

`drm_property_add_enum` — add a possible value to an enumeration property

Synopsis

```
int drm_property_add_enum (struct drm_property * property, int index,  
uint64_t value, const char * name);
```

Arguments

<i>property</i>	enumeration property to change
<i>index</i>	index of the new enumeration
<i>value</i>	value of the new enumeration
<i>name</i>	symbolic name of the new enumeration

Description

This functions adds enumerations to a property.

It's use is deprecated, drivers should use one of the more specific helpers to directly create the property with all enumerations already attached.

Returns

Zero on success, error code on failure.

Name

`drm_property_destroy` — destroy a drm property

Synopsis

```
void drm_property_destroy (struct drm_device * dev, struct drm_property  
* property);
```

Arguments

dev drm device

property property to destroy

Description

This function frees a property including any attached resources like enumeration values.

Name

`drm_object_attach_property` — attach a property to a modeset object

Synopsis

```
void drm_object_attach_property (struct drm_mode_object * obj, struct  
drm_property * property, uint64_t init_val);
```

Arguments

obj drm modeset object

property property to attach

init_val initial value of the property

Description

This attaches the given property to the modeset object with the given initial value. Currently this function cannot fail since the properties are stored in a statically sized array.

Name

`drm_object_property_set_value` — set the value of a property

Synopsis

```
int drm_object_property_set_value (struct drm_mode_object * obj, struct
drm_property * property, uint64_t val);
```

Arguments

obj drm mode object to set property value for

property property to set

val value the property should be set to

Description

This functions sets a given property on a given object. This function only changes the software state of the property, it does not call into the driver's `->set_property` callback.

Returns

Zero on success, error code on failure.

Name

`drm_object_property_get_value` — retrieve the value of a property

Synopsis

```
int drm_object_property_get_value (struct drm_mode_object * obj, struct
drm_property * property, uint64_t * val);
```

Arguments

obj drm mode object to get property value from

property property to retrieve

val storage for the property value

Description

This function retrieves the software state of the given property for the given property. Since there is no driver callback to retrieve the current property value this might be out of sync with the hardware, depending upon the driver and property.

Returns

Zero on success, error code on failure.

Name

`drm_mode_connector_update_edid_property` — update the edid property of a connector

Synopsis

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

Arguments

connector drm connector

edid new value of the edid property

Description

This function creates a new blob modeset object and assigns its id to the connector's edid property.

Returns

Zero on success, errno on failure.

Name

`drm_mode_connector_attach_encoder` — attach a connector to an encoder

Synopsis

```
int    drm_mode_connector_attach_encoder    (struct    drm_connector    *  
connector, struct drm_encoder * encoder);
```

Arguments

connector connector to attach

encoder encoder to attach *connector* to

Description

This function links up a connector to an encoder. Note that the routing restrictions between encoders and crtcs are exposed to userspace through the `possible_clones` and `possible_crtcs` bitmasks.

Returns

Zero on success, `errno` on failure.

Name

`drm_mode_crtc_set_gamma_size` — set the gamma table size

Synopsis

```
int drm_mode_crtc_set_gamma_size (struct drm_crtc * crtc, int
gamma_size);
```

Arguments

crtc CRTC to set the gamma table size for

gamma_size size of the gamma table

Description

Drivers which support gamma tables should set this to the supported gamma table size when initializing the CRTC. Currently the drm core only supports a fixed gamma table size.

Returns

Zero on success, errno on failure.

Name

`drm_mode_config_reset` — call `->reset` callbacks

Synopsis

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

Arguments

dev drm device

Description

This functions calls all the crtcs, encoder's and connector's `->reset` callback. Drivers can use this in e.g. their driver load or resume code to reset hardware and software state.

Name

`drm_fb_get_bpp_depth` — get the bpp/depth values for format

Synopsis

```
void drm_fb_get_bpp_depth (uint32_t format, unsigned int * depth, int  
* bpp);
```

Arguments

format pixel format (DRM_FORMAT_*)

depth storage for the depth value

bpp storage for the bpp value

Description

This only supports RGB formats here for compat with code that doesn't use pixel formats directly yet.

Name

`drm_format_num_planes` — get the number of planes for format

Synopsis

```
int drm_format_num_planes (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 drm_format_plane_cpp (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 drm_format_horz_chroma_subsampling (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 drm_format_vert_chroma_subsampling (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 drm_mode_config_init (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 drm_mode_config_cleanup (struct drm_device * dev);
```

Arguments

dev DRM device

Description

Free up all the connectors and CRTC's 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

KMS Locking

As KMS moves toward more fine grained locking, and atomic ioctl where userspace can indirectly control locking order, it becomes necessary to use `ww_mutex` and acquire-contexts to avoid deadlocks. But because the locking is more distributed around the driver code, we want a bit of extra utility/tracking out of our acquire-ctx. This is provided by `drm_modeset_lock` / `drm_modeset_acquire_ctx`.

For basic principles of `ww_mutex`, see: [Documentation/ww-mutex-design.txt](#)

The basic usage pattern is to:

```
drm_modeset_acquire_init(ctx) retry: foreach (lock in random_ordered_set_of_locks) { ret =  
drm_modeset_lock(lock, ctx) if (ret == -EDEADLK) { drm_modeset_backoff(ctx); goto retry; } }
```

... do stuff ...

```
drm_modeset_drop_locks(ctx); drm_modeset_acquire_fini(ctx);
```

Name

`drm_modeset_lock_init` — initialize lock

Synopsis

```
void drm_modeset_lock_init (struct drm_modeset_lock * lock);
```

Arguments

lock lock to init

Name

`drm_modeset_lock_fini` — cleanup lock

Synopsis

```
void drm_modeset_lock_fini (struct drm_modeset_lock * lock);
```

Arguments

lock lock to cleanup

Name

`drm_modeset_is_locked` — equivalent to `mutex_is_locked`

Synopsis

```
bool drm_modeset_is_locked (struct drm_modeset_lock * lock);
```

Arguments

lock lock to check

Name

`drm_modeset_acquire_init` — initialize acquire context

Synopsis

```
void drm_modeset_acquire_init (struct drm_modeset_acquire_ctx * ctx,  
uint32_t flags);
```

Arguments

ctx the acquire context

flags for future

Name

`drm_modeset_acquire_fini` — cleanup acquire context

Synopsis

```
void drm_modeset_acquire_fini (struct drm_modeset_acquire_ctx * ctx);
```

Arguments

ctx the acquire context

Name

`drm_modeset_drop_locks` — drop all locks

Synopsis

```
void drm_modeset_drop_locks (struct drm_modeset_acquire_ctx * ctx);
```

Arguments

ctx the acquire context

Description

Drop all locks currently held against this acquire context.

Name

`drm_modeset_backoff` — deadlock avoidance backoff

Synopsis

```
void drm_modeset_backoff (struct drm_modeset_acquire_ctx * ctx);
```

Arguments

ctx the acquire context

Description

If deadlock is detected (ie. `drm_modeset_lock` returns `-EDEADLK`), you must call this function to drop all currently held locks and block until the contended lock becomes available.

Name

`drm_modeset_backoff_interruptible` — deadlock avoidance backoff

Synopsis

```
int drm_modeset_backoff_interruptible (struct drm_modeset_acquire_ctx
* ctx);
```

Arguments

ctx the acquire context

Description

Interruptible version of `drm_modeset_backoff`

Name

`drm_modeset_lock` — take modeset lock

Synopsis

```
int drm_modeset_lock (struct drm_modeset_lock * lock, struct
drm_modeset_acquire_ctx * ctx);
```

Arguments

lock lock to take

ctx acquire ctx

Description

If `ctx` is not `NULL`, then its ww acquire context is used and the lock will be tracked by the context and can be released by calling `drm_modeset_drop_locks`. If `-EDEADLK` is returned, this means a deadlock scenario has been detected and it is an error to attempt to take any more locks without first calling `drm_modeset_backoff`.

Name

`drm_modeset_lock_interruptible` — take modeset lock

Synopsis

```
int drm_modeset_lock_interruptible (struct drm_modeset_lock * lock,  
struct drm_modeset_acquire_ctx * ctx);
```

Arguments

lock lock to take

ctx acquire ctx

Description

Interruptible version of `drm_modeset_lock`

Name

`drm_modeset_unlock` — drop modeset lock

Synopsis

```
void drm_modeset_unlock (struct drm_modeset_lock * lock);
```

Arguments

lock lock to release

Mode Setting Helper Functions

The plane, 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 plane, 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 CRTC and encoders to apply the DPMS state to the device.

The mid-layer doesn't track the power state of CRTC 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 optional 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 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 not used?

- The preferred mode for the connector

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).

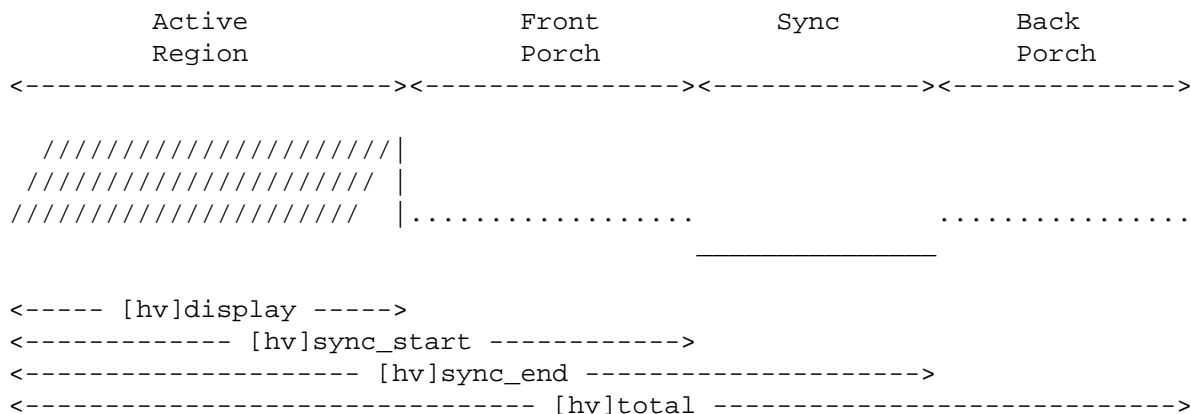
Drivers must set the DRM_MODE_TYPE_DRIVER bit for all modes they create, and set the DRM_MODE_TYPE_PREFERRED bit for the preferred mode.

- `__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 optional.

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 drm_helper_move_panel_connectors_to_head (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_encoder_in_use` — check if a given encoder is in use

Synopsis

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

Arguments

encoder encoder to check

Description

Checks whether *encoder* is with the current mode setting output configuration in use by any connector. This doesn't mean that it is actually enabled since the DPMS state is tracked separately.

Returns

True if *encoder* is used, false otherwise.

Name

`drm_helper_crtc_in_use` — check if a given CRTC is in a mode_config

Synopsis

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

Arguments

crtc CRTC to check

Description

Checks whether *crtc* is with the current mode setting output configuration in use by any connector. This doesn't mean that it is actually enabled since the DPMS state is tracked separately.

Returns

True if *crtc* is used, false otherwise.

Name

`drm_helper_disable_unused_functions` — disable unused objects

Synopsis

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

Arguments

dev DRM device

Description

This function walks through the entire mode setting configuration of *dev*. It will remove any crtc links of unused encoders and encoder links of disconnected connectors. Then it will disable all unused encoders and crtcs either by calling their disable callback if available or by calling their dpms callback with `DRM_MODE_DPMS_OFF`.

Name

`drm_crtc_helper_set_mode` — internal helper to set a mode

Synopsis

```
bool  drm_crtc_helper_set_mode (struct  drm_crtc  *  crtc,  struct
drm_display_mode * mode, int x, int y, struct  drm_framebuffer * old_fb);
```

Arguments

<i>crtc</i>	CRTC to program
<i>mode</i>	mode to use
<i>x</i>	horizontal offset into the surface
<i>y</i>	vertical offset into the surface
<i>old_fb</i>	old framebuffer, for cleanup

Description

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, false otherwise.

Name

`drm_crtc_helper_set_config` — set a new config from userspace

Synopsis

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

Arguments

set mode set configuration

Description

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, negative errno numbers on failure.

Name

`drm_helper_connector_dpms` — connector dpms helper implementation

Synopsis

```
void drm_helper_connector_dpms (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.

Name

`drm_helper_mode_fill_fb_struct` — fill out framebuffer metadata

Synopsis

```
void drm_helper_mode_fill_fb_struct (struct drm_framebuffer * fb, struct
drm_mode_fb_cmd2 * mode_cmd);
```

Arguments

fb drm_framebuffer object to fill out

mode_cmd metadata from the userspace fb creation request

Description

This helper can be used in a drivers `fb_create` callback to pre-fill the fb's metadata fields.

Name

`drm_helper_resume_force_mode` — force-restore mode setting configuration

Synopsis

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

Arguments

dev `drm_device` which should be restored

Description

Drivers which use the mode setting helpers can use this function to force-restore the mode setting configuration e.g. on resume or when something else might have trampled over the hw state (like some overzealous old BIOSen tended to do).

This helper doesn't provide a error return value since restoring the old config should never fail due to resource allocation issues since the driver has successfully set the restored configuration already. Hence this should boil down to the equivalent of a few `dpms` on calls, which also don't provide an error code.

Drivers where simply restoring an old configuration again might fail (e.g. due to slight differences in allocating shared resources when the configuration is restored in a different order than when userspace set it up) need to use their own restore logic.

Output Probing Helper Functions Reference

This library provides some helper code for output probing. It provides an implementation of the `core connector->fill_modes` interface with `drm_helper_probe_single_connector_modes`.

It also provides support for polling connectors with a work item and for generic hotplug interrupt handling where the driver doesn't or cannot keep track of a per-connector hpd interrupt.

This helper library can be used independently of the modeset helper library. Drivers can also overwrite different parts e.g. use their own hotplug handling code to avoid probing unrelated outputs.

Name

`drm_helper_probe_single_connector_modes` — get complete set of display modes

Synopsis

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

Arguments

<i>connector</i>	connector to probe
<i>maxX</i>	max width for modes
<i>maxY</i>	max height for modes

Description

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 `->fill_modes connector` vfunc for drivers that use the `crtc` helpers for output mode filtering and detection.

Returns

The number of modes found on *connector*.

Name

`drm_helper_probe_single_connector_modes_nomerge` — get complete set of display modes

Synopsis

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

Arguments

<i>connector</i>	connector to probe
<i>maxX</i>	max width for modes
<i>maxY</i>	max height for modes

Description

This operates like `drm_helper_probe_single_connector_modes` except it replaces the mode bits instead of merging them for preferred modes.

Name

`drm_kms_helper_hotplug_event` — fire off KMS hotplug events

Synopsis

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

Arguments

dev `drm_device` whose connector state changed

Description

This function fires off the uevent for userspace and also calls the `output_poll_changed` function, which is most commonly used to inform the fbdev emulation code and allow it to update the fbcon output configuration.

Drivers should call this from their hotplug handling code when a change is detected. Note that this function does not do any output detection of its own, like `drm_helper_hpd_irq_event` does - this is assumed to be done by the driver already.

This function must be called from process context with no mode setting locks held.

Name

`drm_kms_helper_poll_disable` — disable output polling

Synopsis

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

Arguments

dev `drm_device`

Description

This function disables the output polling work.

Drivers can call this helper from their device suspend implementation. It is not an error to call this even when output polling isn't enabled or already disabled.

Name

`drm_kms_helper_poll_enable` — re-enable output polling.

Synopsis

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

Arguments

dev `drm_device`

Description

This function re-enables the output polling work.

Drivers can call this helper from their device resume implementation. It is an error to call this when the output polling support has not yet been set up.

Name

`drm_kms_helper_poll_init` — initialize and enable output polling

Synopsis

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

Arguments

dev `drm_device`

Description

This function initializes and then also enables output polling support for *dev*. Drivers which do not have reliable hotplug support in hardware can use this helper infrastructure to regularly poll such connectors for changes in their connection state.

Drivers can control which connectors are polled by setting the `DRM_CONNECTOR_POLL_CONNECT` and `DRM_CONNECTOR_POLL_DISCONNECT` flags. On connectors where probing live outputs can result in visual distortion drivers should not set the `DRM_CONNECTOR_POLL_DISCONNECT` flag to avoid this. Connectors which have no flag or only `DRM_CONNECTOR_POLL_HPD` set are completely ignored by the polling logic.

Note that a connector can be both polled and probed from the hotplug handler, in case the hotplug interrupt is known to be unreliable.

Name

`drm_kms_helper_poll_fini` — disable output polling and clean it up

Synopsis

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

Arguments

dev `drm_device`

Name

`drm_helper_hpd_irq_event` — hotplug processing

Synopsis

```
bool drm_helper_hpd_irq_event (struct drm_device * dev);
```

Arguments

dev `drm_device`

Description

Drivers can use this helper function to run a detect cycle on all connectors which have the `DRM_CONNECTOR_POLL_HPD` flag set in their polled member. All other connectors are ignored, which is useful to avoid reprobing fixed panels.

This helper function is useful for drivers which can't or don't track hotplug interrupts for each connector.

Drivers which support hotplug interrupts for each connector individually and which have a more fine-grained detect logic should bypass this code and directly call `drm_kms_helper_hotplug_event` in case the connector state changed.

This function must be called from process context with no mode setting locks held.

Note that a connector can be both polled and probed from the hotplug handler, in case the hotplug interrupt is known to be unreliable.

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 independently from the crtcs 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 drm_fb_helper_single_add_all_connectors (struct drm_fb_helper *  
fb_helper);
```

Arguments

fb_helper fbdev initialized with `drm_fb_helper_init`

Description

This function 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 drm_fb_helper_debug_enter (struct fb_info * info);
```

Arguments

info fbdev registered by the helper

Name

`drm_fb_helper_debug_leave` — implementation for `->fb_debug_leave`

Synopsis

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

Arguments

info fbdev registered by the helper

Name

`drm_fb_helper_restore_fbdev_mode_unlocked` — restore fbdev configuration

Synopsis

```
bool drm_fb_helper_restore_fbdev_mode_unlocked (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 drm_fb_helper_blank (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 drm_fb_helper_init (struct drm_device * dev, struct drm_fb_helper
* fb_helper, int crtc_count, int max_conn_count);
```

Arguments

<i>dev</i>	drm device
<i>fb_helper</i>	driver-allocated fbdev helper structure to initialize
<i>crtc_count</i>	maximum number of crtcs to support in this fbdev emulation
<i>max_conn_count</i>	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 drm_fb_helper_setcmap (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 drm_fb_helper_check_var (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 drm_fb_helper_set_par (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 drm_fb_helper_pan_display (struct fb_var_screeninfo * var, struct
fb_info * info);
```

Arguments

var updated screen information

info fbdev registered by the helper

Name

`drm_fb_helper_fill_fix` — initializes fixed fbdev information

Synopsis

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

Arguments

info fbdev registered by the helper

pitch desired pitch

depth 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` — initializes variable fbdev information

Synopsis

```
void drm_fb_helper_fill_var (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 drm_fb_helper_initial_config (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 CRTC's 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 drm_fb_helper_hotplug_event (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 regn  
    void (* gamma_get) (struct drm_crtc *crtc, u16 *red, u16 *green, u16 *blue, int r  
    int (* fb_probe) (struct drm_fb_helper *helper, struct drm_fb_helper_surface_size  
    bool (* initial_config) (struct drm_fb_helper *fb_helper, struct drm_fb_helper_cr  
};
```

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. Furthermore 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, ...

The DisplayPort AUX channel is an abstraction to allow generic, driver- independent access to AUX functionality. Drivers can take advantage of this by filling in the fields of the `drm_dp_aux` structure.

Transactions are described using a hardware-independent `drm_dp_aux_msg` structure, which is passed into a driver's `.transfer` implementation. Both native and I2C-over-AUX transactions are supported.

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  
};
```

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

struct `drm_dp_aux_msg` — DisplayPort AUX channel transaction

Synopsis

```
struct drm_dp_aux_msg {  
    unsigned int address;  
    u8 request;  
    u8 reply;  
    void * buffer;  
    size_t size;  
};
```

Members

<code>address</code>	address of the (first) register to access
<code>request</code>	contains the type of transaction (see <code>DP_AUX_*</code> macros)
<code>reply</code>	upon completion, contains the reply type of the transaction
<code>buffer</code>	pointer to a transmission or reception buffer
<code>size</code>	size of <i>buffer</i>

Name

struct drm_dp_aux — DisplayPort AUX channel

Synopsis

```
struct drm_dp_aux {
    const char * name;
    struct i2c_adapter ddc;
    struct device * dev;
    struct mutex hw_mutex;
    ssize_t (* transfer) (struct drm_dp_aux *aux, struct drm_dp_aux_msg *msg);
};
```

Members

name	user-visible name of this AUX channel and the I2C-over-AUX adapter
ddc	I2C adapter that can be used for I2C-over-AUX communication
dev	pointer to struct device that is the parent for this AUX channel
hw_mutex	internal mutex used for locking transfers
transfer	transfers a message representing a single AUX transaction

Description

The `.dev` field should be set to a pointer to the device that implements the AUX channel.

The `.name` field may be used to specify the name of the I2C adapter. If set to NULL, `dev_name` of `.dev` will be used.

Drivers provide a hardware-specific implementation of how transactions are executed via the `.transfer` function. A pointer to a `drm_dp_aux_msg` structure describing the transaction is passed into this function. Upon success, the implementation should return the number of payload bytes that were transferred, or a negative error-code on failure. Helpers propagate errors from the `.transfer` function, with the exception of the `-EBUSY` error, which causes a transaction to be retried. On a short, helpers will return `-EPROTO` to make it simpler to check for failure.

An AUX channel can also be used to transport I2C messages to a sink. A typical application of that is to access an EDID that's present in the sink device. The `.transfer` function can also be used to execute such transactions. The `drm_dp_aux_register_i2c_bus` function registers an I2C adapter that can be passed to `drm_probe_ddc`. Upon removal, drivers should call `drm_dp_aux_unregister_i2c_bus` to remove the I2C adapter.

Note that the aux helper code assumes that the `.transfer` function only modifies the reply field of the `drm_dp_aux_msg` structure. The retry logic and i2c helpers assume this is the case.

Name

`drm_dp_dpcd_readb` — read a single byte from the DPCD

Synopsis

```
ssize_t drm_dp_dpcd_readb (struct drm_dp_aux * aux, unsigned int offset,  
u8 * valuep);
```

Arguments

aux DisplayPort AUX channel

offset address of the register to read

valuep location where the value of the register will be stored

Description

Returns the number of bytes transferred (1) on success, or a negative error code on failure.

Name

`drm_dp_dpcd_writeb` — write a single byte to the DPCD

Synopsis

```
ssize_t drm_dp_dpcd_writeb (struct drm_dp_aux * aux, unsigned int  
offset, u8 value);
```

Arguments

aux DisplayPort AUX channel

offset address of the register to write

value value to write to the register

Description

Returns the number of bytes transferred (1) on success, or a negative error code on failure.

Name

`i2c_dp_aux_add_bus` — register an i2c adapter using the aux ch helper

Synopsis

```
int i2c_dp_aux_add_bus (struct i2c_adapter * adapter);
```

Arguments

adapter i2c adapter to register

Description

This registers an i2c adapter that uses dp aux channel as it's underlying 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.

IMPORTANT

This interface is deprecated, please switch to the new dp aux helpers and `drm_dp_aux_register`.

Name

`drm_dp_dpcd_read` — read a series of bytes from the DPCD

Synopsis

```
ssize_t drm_dp_dpcd_read (struct drm_dp_aux * aux, unsigned int offset,  
void * buffer, size_t size);
```

Arguments

<i>aux</i>	DisplayPort AUX channel
<i>offset</i>	address of the (first) register to read
<i>buffer</i>	buffer to store the register values
<i>size</i>	number of bytes in <i>buffer</i>

Description

Returns the number of bytes transferred on success, or a negative error code on failure. -EIO is returned if the request was NAKed by the sink or if the retry count was exceeded. If not all bytes were transferred, this function returns -EPROTO. Errors from the underlying AUX channel transfer function, with the exception of -EBUSY (which causes the transaction to be retried), are propagated to the caller.

Name

`drm_dp_dpcd_write` — write a series of bytes to the DPCD

Synopsis

```
ssize_t drm_dp_dpcd_write (struct drm_dp_aux * aux, unsigned int offset,  
void * buffer, size_t size);
```

Arguments

<i>aux</i>	DisplayPort AUX channel
<i>offset</i>	address of the (first) register to write
<i>buffer</i>	buffer containing the values to write
<i>size</i>	number of bytes in <i>buffer</i>

Description

Returns the number of bytes transferred on success, or a negative error code on failure. -EIO is returned if the request was NAKed by the sink or if the retry count was exceeded. If not all bytes were transferred, this function returns -EPROTO. Errors from the underlying AUX channel transfer function, with the exception of -EBUSY (which causes the transaction to be retried), are propagated to the caller.

Name

`drm_dp_dpcd_read_link_status` — read DPCD link status (bytes 0x202-0x207)

Synopsis

```
int  drm_dp_dpcd_read_link_status (struct  drm_dp_aux  *  aux,  u8
    status[DP_LINK_STATUS_SIZE]);
```

Arguments

aux DisplayPort AUX channel

status[DP_LINK_STATUS_SIZE] Buffer to store the link status in (must be at least 6 bytes)

Description

Returns the number of bytes transferred on success or a negative error code on failure.

Name

`drm_dp_link_probe` — probe a DisplayPort link for capabilities

Synopsis

```
int drm_dp_link_probe (struct drm_dp_aux * aux, struct drm_dp_link *  
link);
```

Arguments

aux DisplayPort AUX channel

link pointer to structure in which to return link capabilities

Description

The structure filled in by this function can usually be passed directly into `drm_dp_link_power_up` and `drm_dp_link_configure` to power up and configure the link based on the link's capabilities.

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

Name

`drm_dp_link_power_up` — power up a DisplayPort link

Synopsis

```
int drm_dp_link_power_up (struct drm_dp_aux * aux, struct drm_dp_link  
* link);
```

Arguments

aux DisplayPort AUX channel

link pointer to a structure containing the link configuration

Description

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

Name

`drm_dp_link_configure` — configure a DisplayPort link

Synopsis

```
int drm_dp_link_configure (struct drm_dp_aux * aux, struct drm_dp_link  
* link);
```

Arguments

aux DisplayPort AUX channel

link pointer to a structure containing the link configuration

Description

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

Name

`drm_dp_aux_register` — initialise and register aux channel

Synopsis

```
int drm_dp_aux_register (struct drm_dp_aux * aux);
```

Arguments

aux DisplayPort AUX channel

Description

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

Name

`drm_dp_aux_unregister` — unregister an AUX adapter

Synopsis

```
void drm_dp_aux_unregister (struct drm_dp_aux * aux);
```

Arguments

aux DisplayPort AUX channel

EDID Helper Functions Reference

Name

`drm_edid_header_is_valid` — sanity check the header of the base EDID block

Synopsis

```
int drm_edid_header_is_valid (const u8 * raw_edid);
```

Arguments

raw_edid pointer to raw base EDID block

Description

Sanity check the header of the base EDID block.

Return

8 if the header is perfect, down to 0 if it's totally wrong.

Name

`drm_edid_block_valid` — Sanity check the EDID block (base or extension)

Synopsis

```
bool    drm_edid_block_valid    (u8    *    raw_edid,    int    block,    bool  
print_bad_edid);
```

Arguments

raw_edid pointer to raw EDID block

block type of block to validate (0 for base, extension otherwise)

print_bad_edid if true, dump bad EDID blocks to the console

Description

Validate a base or extension EDID block and optionally dump bad blocks to the console.

Return

True if the block is valid, false otherwise.

Name

`drm_edid_is_valid` — sanity check EDID data

Synopsis

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

Arguments

edid EDID data

Description

Sanity-check an entire EDID record (including extensions)

Return

True if the EDID data is valid, false otherwise.

Name

`drm_probe_ddc` — probe DDC presence

Synopsis

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

Arguments

adapter I2C adapter to probe

Return

True on success, false on failure.

Name

`drm_get_edid` — get EDID data, if available

Synopsis

```
struct edid * drm_get_edid (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

Pointer to valid EDID or NULL if we couldn't find any.

Name

`drm_edid_duplicate` — duplicate an EDID and the extensions

Synopsis

```
struct edid * drm_edid_duplicate (const struct edid * edid);
```

Arguments

edid EDID to duplicate

Return

Pointer to duplicated EDID or NULL on allocation failure.

Name

`drm_match_cea_mode` — look for a CEA mode matching given mode

Synopsis

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

Arguments

to_match display mode

Return

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

Name

`drm_get_cea_aspect_ratio` — get the picture aspect ratio corresponding to the input VIC from the CEA mode list

Synopsis

```
enum    hdmi_picture_aspect    drm_get_cea_aspect_ratio    (const    u8
video_code);
```

Arguments

video_code ID given to each of the CEA modes

Description

Returns picture aspect ratio

Name

`drm_edid_to_eld` — build ELD from EDID

Synopsis

```
void drm_edid_to_eld (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. The `Conn_Type`, `HDCP` and `Port_ID` ELD fields are left for the graphics driver to fill in.

Name

`drm_edid_to_sad` — extracts SADs from EDID

Synopsis

```
int drm_edid_to_sad (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

The returned pointer needs to be freed using `kfree`.

Return

The number of found SADs or negative number on error.

Name

`drm_edid_to_speaker_allocation` — extracts Speaker Allocation Data Blocks from EDID

Synopsis

```
int drm_edid_to_speaker_allocation (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

The returned pointer needs to be freed using `kfree`.

Return

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

Name

`drm_av_sync_delay` — compute the HDMI/DP sink audio-video sync delay

Synopsis

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

Arguments

connector connector associated with the HDMI/DP sink

mode the display mode

Return

The HDMI/DP sink's audio-video sync delay in milliseconds or 0 if the sink doesn't support audio or video.

Name

`drm_select_eld` — select one ELD from multiple HDMI/DP sinks

Synopsis

```
struct drm_connector * drm_select_eld (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.

Return

The connector associated with the first HDMI/DP sink that has ELD attached to it.

Name

`drm_detect_hdmi_monitor` — detect whether monitor is HDMI

Synopsis

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

Arguments

edid monitor EDID information

Description

Parse the CEA extension according to CEA-861-B.

Return

True if the monitor is HDMI, false if not or unknown.

Name

`drm_detect_monitor_audio` — check monitor audio capability

Synopsis

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

Arguments

edid EDID block to scan

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.

Return

True if the monitor supports audio, false otherwise.

Name

`drm_rgb_quant_range_selectable` — is RGB quantization range selectable?

Synopsis

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

Arguments

edid EDID block to scan

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.

Return

True if the RGB quantization range is selectable, false otherwise.

Name

`drm_add_edid_modes` — add modes from EDID data, if available

Synopsis

```
int drm_add_edid_modes (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

The 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  drm_add_modes_noedid (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

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

Name

`drm_set_preferred_mode` — Sets the preferred mode of a connector

Synopsis

```
void drm_set_preferred_mode (struct drm_connector * connector, int  
hpref, int vpref);
```

Arguments

connector connector whose mode list should be processed

hpref horizontal resolution of preferred mode

vpref vertical resolution of preferred mode

Description

Marks a mode as preferred if it matches the resolution specified by *hpref* and *vpref*.

Name

`drm_hdmi_avi_infoframe_from_display_mode` — fill an HDMI AVI infoframe with data from a DRM display mode

Synopsis

```
int drm_hdmi_avi_infoframe_from_display_mode (struct hdmi_avi_infoframe
* frame, const struct drm_display_mode * mode);
```

Arguments

frame HDMI AVI infoframe

mode DRM display mode

Return

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

Arguments

frame HDMI vendor infoframe

mode 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.

Return

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)

Name

`drm_rect_adjust_size` — adjust the size of the rectangle

Synopsis

```
void drm_rect_adjust_size (struct drm_rect * r, int dw, int dh);
```

Arguments

r rectangle to be adjusted

dw horizontal adjustment

dh vertical adjustment

Description

Change the size of rectangle *r* by *dw* in the horizontal direction, and by *dh* in the vertical direction, while keeping the center of *r* stationary.

Positive *dw* and *dh* increase the size, negative values decrease it.

Name

`drm_rect_translate` — translate the rectangle

Synopsis

```
void drm_rect_translate (struct drm_rect * r, int dx, int dy);
```

Arguments

r rectangle to be translated

dx horizontal translation

dy vertical translation

Description

Move rectangle *r* by *dx* in the horizontal direction, and by *dy* in the vertical direction.

Name

`drm_rect_downscale` — downscale a rectangle

Synopsis

```
void drm_rect_downscale (struct drm_rect * r, 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 drm_rect_width (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 drm_rect_height (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 drm_rect_visible (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 drm_rect_equals (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 drm_rect_intersect (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 drm_rect_clip_scaled (struct drm_rect * src, struct drm_rect * dst,  
const struct drm_rect * clip, int hscale, int vscale);
```

Arguments

<i>src</i>	source window rectangle
<i>dst</i>	destination window rectangle
<i>clip</i>	clip rectangle
<i>hscale</i>	horizontal scaling factor
<i>vscale</i>	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 drm_rect_calc_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 $(src \text{ width}) / (dst \text{ width})$.

RETURNS

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

Name

`drm_rect_calc_vscale` — calculate the vertical scaling factor

Synopsis

```
int drm_rect_calc_vscale (const struct drm_rect * src, const struct
drm_rect * dst, int min_vscale, int max_vscale);
```

Arguments

<i>src</i>	source window rectangle
<i>dst</i>	destination window rectangle
<i>min_vscale</i>	minimum allowed vertical scaling factor
<i>max_vscale</i>	maximum allowed vertical scaling factor

Description

Calculate the vertical scaling factor as $(src \text{ height}) / (dst \text{ height})$.

RETURNS

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

Name

`drm_rect_calc_hscale_relaxed` — calculate the horizontal scaling factor

Synopsis

```
int drm_rect_calc_hscale_relaxed (struct drm_rect * src, struct drm_rect  
* dst, int min_hscale, int max_hscale);
```

Arguments

<i>src</i>	source window rectangle
<i>dst</i>	destination window rectangle
<i>min_hscale</i>	minimum allowed horizontal scaling factor
<i>max_hscale</i>	maximum allowed horizontal scaling factor

Description

Calculate the horizontal scaling factor as $(src \text{ width}) / (dst \text{ width})$.

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 drm_rect_calc_vscale_relaxed (struct drm_rect * src, struct drm_rect  
* dst, int min_vscale, int max_vscale);
```

Arguments

<i>src</i>	source window rectangle
<i>dst</i>	destination window rectangle
<i>min_vscale</i>	minimum allowed vertical scaling factor
<i>max_vscale</i>	maximum allowed vertical scaling factor

Description

Calculate the vertical scaling factor as $(src \text{ height}) / (dst \text{ height})$.

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 drm_rect_debug_print (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 unref 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

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

Name

`drm_flip_work_queue` — queue work

Synopsis

```
void drm_flip_work_queue (struct drm_flip_work * work, void * val);
```

Arguments

work the flip-work

val 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 drm_flip_work_commit (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 drm_flip_work_init (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 drm_flip_work_cleanup (struct drm_flip_work * work);
```

Arguments

work the flip-work to cleanup

Description

Destroy resources allocated for the flip-work

HDMI Infoframes Helper Reference

Strictly speaking this is not a DRM helper library but generally useable by any driver interfacing with HDMI outputs like v4l or alsa drivers. But it nicely fits into the overall topic of mode setting helper libraries and hence is also included here.

Name

union hdmi_infoframe — overall union of all abstract infoframe representations

Synopsis

```
union hdmi_infoframe {
    struct hdmi_any_infoframe any;
    struct hdmi_avi_infoframe avi;
    struct hdmi_spd_infoframe spd;
    union hdmi_vendor_any_infoframe vendor;
    struct hdmi_audio_infoframe audio;
};
```

Members

any	generic infoframe
avi	avi infoframe
spd	spd infoframe
vendor	union of all vendor infoframes
audio	audio infoframe

Description

This is used by the generic pack function. This works since all infoframes have the same header which also indicates which type of infoframe should be packed.

Name

`hdmi_avi_infoframe_init` — initialize an HDMI AVI infoframe

Synopsis

```
int hdmi_avi_infoframe_init (struct hdmi_avi_infoframe * frame);
```

Arguments

frame HDMI AVI infoframe

Description

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

Name

`hdmi_avi_infoframe_pack` — write HDMI AVI infoframe to binary buffer

Synopsis

```
ssize_t hdmi_avi_infoframe_pack (struct hdmi_avi_infoframe * frame, void  
* buffer, size_t size);
```

Arguments

frame HDMI AVI infoframe

buffer destination buffer

size size of buffer

Description

Packs the information contained in the *frame* structure into a binary representation that can be written into the corresponding controller registers. Also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

Returns the number of bytes packed into the binary buffer or a negative error code on failure.

Name

`hdmi_spd_infoframe_init` — initialize an HDMI SPD infoframe

Synopsis

```
int hdmi_spd_infoframe_init (struct hdmi_spd_infoframe * frame, const  
char * vendor, const char * product);
```

Arguments

frame HDMI SPD infoframe

vendor vendor string

product product string

Description

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

Name

`hdmi_spd_infoframe_pack` — write HDMI SPD infoframe to binary buffer

Synopsis

```
ssize_t hdmi_spd_infoframe_pack (struct hdmi_spd_infoframe * frame, void  
* buffer, size_t size);
```

Arguments

frame HDMI SPD infoframe

buffer destination buffer

size size of buffer

Description

Packs the information contained in the *frame* structure into a binary representation that can be written into the corresponding controller registers. Also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

Returns the number of bytes packed into the binary buffer or a negative error code on failure.

Name

`hdmi_audio_infoframe_init` — initialize an HDMI audio infoframe

Synopsis

```
int hdmi_audio_infoframe_init (struct hdmi_audio_infoframe * frame);
```

Arguments

frame HDMI audio infoframe

Description

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

Name

`hdmi_audio_infoframe_pack` — write HDMI audio infoframe to binary buffer

Synopsis

```
ssize_t hdmi_audio_infoframe_pack (struct hdmi_audio_infoframe * frame,  
void * buffer, size_t size);
```

Arguments

frame HDMI audio infoframe

buffer destination buffer

size size of buffer

Description

Packs the information contained in the *frame* structure into a binary representation that can be written into the corresponding controller registers. Also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

Returns the number of bytes packed into the binary buffer or a negative error code on failure.

Name

`hdmi_vendor_infoframe_init` — initialize an HDMI vendor infoframe

Synopsis

```
int hdmi_vendor_infoframe_init (struct hdmi_vendor_infoframe * frame);
```

Arguments

frame HDMI vendor infoframe

Description

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

Name

`hdmi_vendor_infoframe_pack` — write a HDMI vendor infoframe to binary buffer

Synopsis

```
ssize_t hdmi_vendor_infoframe_pack (struct hdmi_vendor_infoframe *  
frame, void * buffer, size_t size);
```

Arguments

frame HDMI infoframe

buffer destination buffer

size size of buffer

Description

Packs the information contained in the *frame* structure into a binary representation that can be written into the corresponding controller registers. Also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

Returns the number of bytes packed into the binary buffer or a negative error code on failure.

Name

`hdmi_infoframe_pack` — write a HDMI infoframe to binary buffer

Synopsis

```
ssize_t hdmi_infoframe_pack (union hdmi_infoframe * frame, void *  
buffer, size_t size);
```

Arguments

frame HDMI infoframe

buffer destination buffer

size size of buffer

Description

Packs the information contained in the *frame* structure into a binary representation that can be written into the corresponding controller registers. Also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

Returns the number of bytes packed into the binary buffer or a negative error code on failure.

Plane Helper Reference

Name

`drm_plane_helper_check_update` — Check plane update for validity

Synopsis

```
int drm_plane_helper_check_update (struct drm_plane * plane, struct
drm_crtc * crtc, struct drm_framebuffer * fb, struct drm_rect * src,
struct drm_rect * dest, const struct drm_rect * clip, int min_scale, int
max_scale, bool can_position, bool can_update_disabled, bool * visible);
```

Arguments

<i>plane</i>	plane object to update
<i>crtc</i>	owning CRTC of owning plane
<i>fb</i>	framebuffer to flip onto plane
<i>src</i>	source coordinates in 16.16 fixed point
<i>dest</i>	integer destination coordinates
<i>clip</i>	integer clipping coordinates
<i>min_scale</i>	minimum <i>src:dest</i> scaling factor in 16.16 fixed point
<i>max_scale</i>	maximum <i>src:dest</i> scaling factor in 16.16 fixed point
<i>can_position</i>	is it legal to position the plane such that it doesn't cover the entire crtc? This will generally only be false for primary planes.
<i>can_update_disabled</i>	can the plane be updated while the crtc is disabled?
<i>visible</i>	output parameter indicating whether plane is still visible after clipping

Description

Checks that a desired plane update is valid. Drivers that provide their own plane handling rather than helper-provided implementations may still wish to call this function to avoid duplication of error checking code.

RETURNS

Zero if update appears valid, error code on failure

Name

`drm_primary_helper_update` — Helper for primary plane update

Synopsis

```
int drm_primary_helper_update (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);
```

Arguments

<i>plane</i>	plane object to update
<i>crtc</i>	owning CRTC of owning plane
<i>fb</i>	framebuffer to flip onto plane
<i>crtc_x</i>	x offset of primary plane on crtc
<i>crtc_y</i>	y offset of primary plane on crtc
<i>crtc_w</i>	width of primary plane rectangle on crtc
<i>crtc_h</i>	height of primary plane rectangle on crtc
<i>src_x</i>	x offset of <i>fb</i> for panning
<i>src_y</i>	y offset of <i>fb</i> for panning
<i>src_w</i>	width of source rectangle in <i>fb</i>
<i>src_h</i>	height of source rectangle in <i>fb</i>

Description

Provides a default plane update handler for primary planes. This handler is called in response to a userspace SetPlane operation on the plane with a non-NULL framebuffer. We call the driver's modeset handler to update the framebuffer.

SetPlane on a primary plane of a disabled CRTC is not supported, and will return an error.

Note that we make some assumptions about hardware limitations that may not be true for all hardware -- 1) Primary plane cannot be repositioned. 2) Primary plane cannot be scaled. 3) Primary plane must cover the entire CRTC. 4) Subpixel positioning is not supported. Drivers for hardware that don't have these restrictions can provide their own implementation rather than using this helper.

RETURNS

Zero on success, error code on failure

Name

`drm_primary_helper_disable` — Helper for primary plane disable

Synopsis

```
int drm_primary_helper_disable (struct drm_plane * plane);
```

Arguments

plane plane to disable

Description

Provides a default plane disable handler for primary planes. This handler is called in response to a userspace SetPlane operation on the plane with a NULL framebuffer parameter. It unconditionally fails the disable call with -EINVAL the only way to disable the primary plane without driver support is to disable the entire CRTC. Which does not match the plane ->disable hook.

Note that some hardware may be able to disable the primary plane without disabling the whole CRTC. Drivers for such hardware should provide their own disable handler that disables just the primary plane (and they'll likely need to provide their own update handler as well to properly re-enable a disabled primary plane).

RETURNS

Unconditionally returns -EINVAL.

Name

`drm_primary_helper_destroy` — Helper for primary plane destruction

Synopsis

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

Arguments

plane plane to destroy

Description

Provides a default plane destroy handler for primary planes. This handler is called during CRTC destruction. We disable the primary plane, remove it from the DRM plane list, and deallocate the plane structure.

Name

`drm_primary_helper_create_plane` — Create a generic primary plane

Synopsis

```
struct drm_plane * drm_primary_helper_create_plane (struct drm_device  
* dev, const uint32_t * formats, int num_formats);
```

Arguments

<i>dev</i>	drm device
<i>formats</i>	pixel formats supported, or NULL for a default safe list
<i>num_formats</i>	size of <i>formats</i> ; ignored if <i>formats</i> is NULL

Description

Allocates and initializes a primary plane that can be used with the primary plane helpers. Drivers that wish to use driver-specific plane structures or provide custom handler functions may perform their own allocation and initialization rather than calling this function.

Name

`drm_crtc_init` — Legacy CRTC initialization function

Synopsis

```
int drm_crtc_init (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

Initialize a CRTC object with a default helper-provided primary plane and no cursor plane.

Returns

Zero on success, error code on failure.

KMS Properties

Drivers may need to expose additional parameters to applications than those described in the previous sections. KMS supports attaching properties to CRTCs, 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.

Existing KMS Properties

The following table gives description of drm properties exposed by various modules/drivers.

Table 2.1.

Owner Module/ Drivers	Group	Property Name	Type	Property Values	Object attached	Description/ Restrictions
DRM	Generic	“EDID”	BLOB IMMUTABLE	0	Connector	Contains id of edid blob ptr object.

	“DPMS”	ENUM	{ “On”, “Standby”, “Suspend”, “Off” }	Connector	Contains DPMS operation mode value.
Plane	“type”	ENUM IMMUTABLE	{ “Overlay”, “Primary”, “Cursor” }	Plane	Plane type
DVI-I	“subconnector”	ENUM	{ “Unknown”, “DVI-D”, “DVI-A” }	Connector	TBD
	“select subconnector”	ENUM	{ “Automatic”, “DVI-D”, “DVI-A” }	Connector	TBD
TV	“subconnector”	ENUM	{ “Unknown”, “Composite”, “SVIDEO”, “Component”, “SCART” }	Connector	TBD
	“select subconnector”	ENUM	{ “Automatic”, “Composite”, “SVIDEO”, “Component”, “SCART” }	Connector	TBD
	“mode”	ENUM	{ “NTSC_M”, “NTSC_J”, “NTSC_443”, “PAL_B” } etc.	Connector	TBD
	“left margin”	RANGE	Min=0, Max=100	Connector	TBD
	“right margin”	RANGE	Min=0, Max=100	Connector	TBD
	“top margin”	RANGE	Min=0, Max=100	Connector	TBD
	“bottom margin”	RANGE	Min=0, Max=100	Connector	TBD
	“brightness”	RANGE	Min=0, Max=100	Connector	TBD
	“contrast”	RANGE	Min=0, Max=100	Connector	TBD
	“flicker reduction”	RANGE	Min=0, Max=100	Connector	TBD
	“overscan”	RANGE	Min=0, Max=100	Connector	TBD
	“saturation”	RANGE	Min=0, Max=100	Connector	TBD

i915	Optional	"hue"	RANGE	Min=0, Max=100	Connector	TBD
		"scaling mode"	ENUM	{ "None", "Full", "Center", "Full aspect" }	Connector	TBD
		"dirty"	ENUM IMMUTABLE	{ "Off", "On", "Annotate" }	Connector	TBD
	Generic	"Broadcast RGB"	ENUM	{ "Automatic", "Full", "Limited 16:235" }	Connector	TBD
		"audio"	ENUM	{ "force-dvi", "off", "auto", "on" }	Connector	TBD
		Standard name as in DRM	Standard type as in DRM	Standard value as in DRM	Standard Object as in DRM	TBD
	SDVO-TV	"mode"	ENUM	{ "NTSC_M", "NTSC_J", "NTSC_443", "PAL_B" } etc.	Connector	TBD
		"left_margin"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"right_margin"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"top_margin"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"bottom_margin"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"hpos"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"vpos"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"contrast"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD

		"saturation"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"hue"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"sharpness"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"flicker_filter"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"flicker_filter_RANGE"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"flicker_filter_RANGE"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"tv_chroma_filter"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"tv_luma_filter"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"dot_crawl"	RANGE	Min=0, Max=1	Connector	TBD
CDV gma-500	SDVO-TV/ LVDS	"brightness"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
	Generic	"Broadcast RGB"	ENUM	{ "Full", "Limited 16:235" }	Connector	TBD
		"Broadcast RGB"	ENUM	{ "off", "auto", "on" }	Connector	TBD
		Standard name as in DRM	Standard type as in DRM	Standard value as in DRM	Standard Object as in DRM	TBD
Poulsbo	Generic	"backlight"	RANGE	Min=0, Max=100	Connector	TBD
		Standard name as in DRM	Standard type as in DRM	Standard value as in DRM	Standard Object as in DRM	TBD
	SDVO-TV	"mode"	ENUM	{ "NTSC_M", "NTSC_J", "NTSC_443", "PAL_B" } etc.	Connector	TBD

		"left_margin"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"right_margin"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"top_margin"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"bottom_margin"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"hpos"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"vpos"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"contrast"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"saturation"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"hue"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"sharpness"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"flicker_filter"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"flicker_filter_range"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"flicker_filter_gain"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"tv_chroma_filter"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"tv_luma_filter"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
		"dot_crawl"	RANGE	Min=0, Max=1	Connector	TBD

	SDVO-TV/ LVDS	"brightness"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
armada	CRTC	"CSC_YUV"	ENUM	{ "Auto" , "CCIR601", "CCIR709" }	CRTC	TBD
		"CSC_RGB"	ENUM	{ "Auto", "Computer system", "Studio" }	CRTC	TBD
	Overlay	"colorkey"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_min"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_max"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_val"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_alpha"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_mode"	ENUM	{ "disabled", "Y component", "U component" , "V component", "RGB", "R component", "G component", "B component" }	Plane	TBD
		"brightness"	RANGE	Min=0, Max=256 + 255	Plane	TBD
		"contrast"	RANGE	Min=0, Max=0x7fff	Plane	TBD
		"saturation"	RANGE	Min=0, Max=0x7fff	Plane	TBD
exynos	CRTC	"mode"	ENUM	{ "normal", "blank" }	CRTC	TBD
	Overlay	"zpos"	RANGE	Min=0, Max=MAX_PLANE-1	Plane	TBD
i2c/ ch7006_drv	Generic	"scale"	RANGE	Min=0, Max=2	Connector	TBD

	TV	Standard names as in DRM	Standard types as in DRM	Standard Values as in DRM	Standard object as in DRM	TBD
		“mode”	ENUM	{ "PAL", "PAL-M", "PAL-N", "PAL-Nc", "PAL-60", "NTSC-M", "NTSC-J" }	Connector	TBD
nouveau	NV10 Overlay	"colorkey"	RANGE	Min=0, Max=0x01ffffff	Plane	TBD
		“contrast”	RANGE	Min=0, Max=8192-1	Plane	TBD
		“brightness”	RANGE	Min=0, Max=1024	Plane	TBD
		“hue”	RANGE	Min=0, Max=359	Plane	TBD
		“saturation”	RANGE	Min=0, Max=8192-1	Plane	TBD
		“iturbt_709”	RANGE	Min=0, Max=1	Plane	TBD
	Nv04 Overlay	“colorkey”	RANGE	Min=0, Max=0x01ffffff	Plane	TBD
		“brightness”	RANGE	Min=0, Max=1024	Plane	TBD
	Display	“dithering mode”	ENUM	{ "auto", "off", "on" }	Connector	TBD
		“dithering depth”	ENUM	{ "auto", "off", "on", "static 2x2", "dynamic 2x2", "temporal" }	Connector	TBD
		“underscan”	ENUM	{ "auto", "6 bpc", "8 bpc" }	Connector	TBD
		“underscan hborder”	RANGE	Min=0, Max=128	Connector	TBD
		“underscan vborder”	RANGE	Min=0, Max=128	Connector	TBD
		“vibrant hue”	RANGE	Min=0, Max=180	Connector	TBD
		“color vibrance”	RANGE	Min=0, Max=200	Connector	TBD

	Generic	Standard name as in DRM	Standard type as in DRM	Standard value as in DRM	Standard Object as in DRM	TBD
omap	Generic	“rotation”	BITMASK	{ 0, "rotate-0" }, { 1, "rotate-90" }, { 2, "rotate-180" }, { 3, "rotate-270" }, { 4, "reflect-x" }, { 5, "reflect-y" }	CRTC, Plane	TBD
		“zorder”	RANGE	Min=0, Max=3	CRTC, Plane	TBD
qxl	Generic	“hotplug_mode_update”	RANGE	Min=0, Max=1	Connector	TBD
radeon	DVI-I	“coherent”	RANGE	Min=0, Max=1	Connector	TBD
	DAC enable load detect	“load detection”	RANGE	Min=0, Max=1	Connector	TBD
	TV Standard	"tv standard"	ENUM	{ "ntsc", "pal", "pal-m", "pal-60", "ntsc-j", "scart-pal", "pal-cn", "secam" }	Connector	TBD
	legacy TMDS PLL detect	"tmds_pll"	ENUM	{ "driver", "bios" }	-	TBD
	Underscan	"underscan"	ENUM	{ "off", "on", "auto" }	Connector	TBD
		"underscan hborder"	RANGE	Min=0, Max=128	Connector	TBD
		"underscan vborder"	RANGE	Min=0, Max=128	Connector	TBD
	Audio	“audio”	ENUM	{ "off", "on", "auto" }	Connector	TBD
	FMT Dithering	“dither”	ENUM	{ "off", "on" }	Connector	TBD
	Generic	Standard name as in DRM	Standard type as in DRM	Standard value as in DRM	Standard Object as in DRM	TBD
rcar-du	Generic	"alpha"	RANGE	Min=0, Max=255	Plane	TBD

		"colorkey"	RANGE	Min=0, Max=0x01ffffff	Plane	TBD
		"zpos"	RANGE	Min=1, Max=7	Plane	TBD

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.

Vertical Blanking and Interrupt Handling Functions Reference

Name

`drm_vblank_cleanup` — cleanup vblank support

Synopsis

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

Arguments

dev DRM device

Description

This function cleans up any resources allocated in `drm_vblank_init`.

Name

`drm_vblank_init` — initialize vblank support

Synopsis

```
int drm_vblank_init (struct drm_device * dev, int num_crtcs);
```

Arguments

dev `drm_device`

num_crtcs number of crtcs supported by *dev*

Description

This function initializes vblank support for *num_crtcs* display pipelines.

Returns

Zero on success or a negative error code on failure.

Name

`drm_irq_install` — install IRQ handler

Synopsis

```
int drm_irq_install (struct drm_device * dev, int irq);
```

Arguments

dev DRM device

irq IRQ number to install the handler for

Description

Initializes the IRQ related data. Installs the handler, calling the driver `irq_preinstall` and `irq_postinstall` functions before and after the installation.

This is the simplified helper interface provided for drivers with no special needs. Drivers which need to install interrupt handlers for multiple interrupts must instead set `drm_device->irq_enabled` to signal the DRM core that vblank interrupts are available.

Returns

Zero on success or a negative error code on failure.

Name

`drm_irq_uninstall` — uninstall the IRQ handler

Synopsis

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

Arguments

dev DRM device

Description

Calls the driver's `irq_uninstall` function and unregisters the IRQ handler. This should only be called by drivers which used `drm_irq_install` to set up their interrupt handler. Other drivers must only reset `drm_device->irq_enabled` to false.

Note that for kernel modesetting drivers it is a bug if this function fails. The sanity checks are only to catch buggy user modesetting drivers which call the same function through an `ioctl`.

Returns

Zero on success or a negative error code on failure.

Name

`drm_calc_timestamping_constants` — calculate vblank timestamp constants

Synopsis

```
void drm_calc_timestamping_constants (struct drm_crtc * crtc, const  
struct drm_display_mode * mode);
```

Arguments

crtc `drm_crtc` whose timestamp constants should be updated.

mode display mode containing the scanout timings

Description

Calculate and store various constants which are later needed by vblank and swap-completion timestamping, e.g, by `drm_calc_vbltimestamp_from_scanoutpos`. They are derived from CRTC's true scanout timing, so they take things like panel scaling or other adjustments into account.

Name

`drm_calc_vbltimestamp_from_scanoutpos` — precise vblank timestamp helper

Synopsis

```
int drm_calc_vbltimestamp_from_scanoutpos (struct drm_device * dev, int
crtc, int * max_error, struct timeval * vblank_time, unsigned flags,
const struct drm_crtc * refcrtc, const struct drm_display_mode * mode);
```

Arguments

<i>dev</i>	DRM device
<i>crtc</i>	Which CRTC's vblank timestamp to retrieve
<i>max_error</i>	Desired maximum allowable error in timestamps (nanosecs) On return contains true maximum error of timestamp
<i>vblank_time</i>	Pointer to struct timeval which should receive the timestamp
<i>flags</i>	Flags to pass to driver: 0 = Default, <code>DRM_CALLED_FROM_VBLIRQ</code> = If function is called from vbl IRQ handler
<i>refcrtc</i>	CRTC which defines scanout timing
<i>mode</i>	mode which defines the scanout timings

Description

Implements calculation of exact vblank timestamps from given `drm_display_mode` timings and current video scanout position of a CRTC. This can be called from within `get_vblank_timestamp` implementation of a kms driver to implement the actual timestamping.

Should return timestamps conforming to the `OML_sync_control` OpenML extension specification. The timestamp corresponds to the end of the vblank interval, aka start of scanout of topmost-leftmost display pixel in the following video frame.

Requires support for optional `dev->driver->get_scanout_position` in kms driver, plus a bit of setup code to provide a `drm_display_mode` that corresponds to the true scanout timing.

The current implementation only handles standard video modes. It returns as no operation if a doublescan or interlaced video mode is active. Higher level code is expected to handle this.

Returns

Negative value on error, failure or if not supported in current

video mode

-EINVAL - Invalid CRTC. -EAGAIN - Temporary unavailable, e.g., called before initial modeset. -ENOTSUPP - Function not supported in current display mode. -EIO - Failed, e.g., due to failed scanout position query.

Returns or'ed positive status flags on success:

DRM_VBLANKTIME_SCANOUTPOS_METHOD - Signal this method used for timestamping.
DRM_VBLANKTIME_INVBL - Timestamp taken while scanout was in vblank interval.

Name

`drm_get_last_vbltimestamp` — retrieve raw timestamp for the most recent vblank interval

Synopsis

```
u32 drm_get_last_vbltimestamp (struct drm_device * dev, int crtc, struct
timeval * tvblank, unsigned flags);
```

Arguments

<i>dev</i>	DRM device
<i>crtc</i>	which CRTC's vblank timestamp to retrieve
<i>tvblank</i>	Pointer to target struct timeval which should receive the timestamp
<i>flags</i>	Flags to pass to driver: 0 = Default, DRM_CALLED_FROM_VBLIRQ = If function is called from vbl IRQ handler

Description

Fetches the system timestamp corresponding to the time of the most recent vblank interval on specified CRTC. May call into kms-driver to compute the timestamp with a high-precision GPU specific method.

Returns zero if timestamp originates from uncorrected `do_gettimeofday` call, i.e., it isn't very precisely locked to the true vblank.

Returns

Non-zero if timestamp is considered to be very precise, zero otherwise.

Name

`drm_vblank_count` — retrieve “cooked” vblank counter value

Synopsis

```
u32 drm_vblank_count (struct drm_device * dev, int crtc);
```

Arguments

dev DRM device

crtc which counter to retrieve

Description

Fetches the “cooked” vblank count value that represents the number of vblank events since the system was booted, including lost events due to modesetting activity.

Returns

The software vblank counter.

Name

`drm_vblank_count_and_time` — retrieve “cooked” vblank counter value and the system timestamp corresponding to that vblank counter value.

Synopsis

```
u32 drm_vblank_count_and_time (struct drm_device * dev, int crtc, struct
timeval * vblanktime);
```

Arguments

<i>dev</i>	DRM device
<i>crtc</i>	which counter to retrieve
<i>vblanktime</i>	Pointer to struct timeval to receive the vblank timestamp.

Description

Fetches the “cooked” vblank count value that represents the number of vblank events since the system was booted, including lost events due to modesetting activity. Returns corresponding system timestamp of the time of the vblank interval that corresponds to the current vblank counter value.

Name

`drm_send_vblank_event` — helper to send vblank event after pageflip

Synopsis

```
void drm_send_vblank_event (struct drm_device * dev, int crtc, struct  
drm_pending_vblank_event * e);
```

Arguments

dev DRM device

crtc CRTC in question

e the event to send

Description

Updates sequence # and timestamp on event, and sends it to userspace. Caller must hold event lock.

Name

`drm_vblank_get` — get a reference count on vblank events

Synopsis

```
int drm_vblank_get (struct drm_device * dev, int crtc);
```

Arguments

dev DRM device

crtc which CRTC to own

Description

Acquire a reference count on vblank events to avoid having them disabled while in use.

This is the legacy version of `drm_crtc_vblank_get`.

Returns

Zero on success, nonzero on failure.

Name

`drm_crtc_vblank_get` — get a reference count on vblank events

Synopsis

```
int drm_crtc_vblank_get (struct drm_crtc * crtc);
```

Arguments

crtc which CRTC to own

Description

Acquire a reference count on vblank events to avoid having them disabled while in use.

This is the native kms version of `drm_vblank_off`.

Returns

Zero on success, nonzero on failure.

Name

`drm_vblank_put` — give up ownership of vblank events

Synopsis

```
void drm_vblank_put (struct drm_device * dev, int crtc);
```

Arguments

dev DRM device

crtc which counter to give up

Description

Release ownership of a given vblank counter, turning off interrupts if possible. Disable interrupts after `drm_vblank_offdelay` milliseconds.

This is the legacy version of `drm_crtc_vblank_put`.

Name

`drm_crtc_vblank_put` — give up ownership of vblank events

Synopsis

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

Arguments

crtc which counter to give up

Description

Release ownership of a given vblank counter, turning off interrupts if possible. Disable interrupts after `drm_vblank_offdelay` milliseconds.

This is the native kms version of `drm_vblank_put`.

Name

`drm_vblank_off` — disable vblank events on a CRTC

Synopsis

```
void drm_vblank_off (struct drm_device * dev, int crtc);
```

Arguments

dev DRM device

crtc CRTC in question

Description

Drivers can use this function to shut down the vblank interrupt handling when disabling a crtc. This function ensures that the latest vblank frame count is stored so that `drm_vblank_on` can restore it again.

Drivers must use this function when the hardware vblank counter can get reset, e.g. when suspending.

This is the legacy version of `drm_crtc_vblank_off`.

Name

`drm_crtc_vblank_off` — disable vblank events on a CRTC

Synopsis

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

Arguments

crtc CRTC in question

Description

Drivers can use this function to shut down the vblank interrupt handling when disabling a crtc. This function ensures that the latest vblank frame count is stored so that `drm_vblank_on` can restore it again.

Drivers must use this function when the hardware vblank counter can get reset, e.g. when suspending.

This is the native kms version of `drm_vblank_off`.

Name

`drm_vblank_on` — enable vblank events on a CRTC

Synopsis

```
void drm_vblank_on (struct drm_device * dev, int crtc);
```

Arguments

dev DRM device

crtc CRTC in question

Description

This functions restores the vblank interrupt state captured with `drm_vblank_off` again. Note that calls to `drm_vblank_on` and `drm_vblank_off` can be unbalanced and so can also be unconditionally called in driver load code to reflect the current hardware state of the crtc.

This is the legacy version of `drm_crtc_vblank_on`.

Name

`drm_crtc_vblank_on` — enable vblank events on a CRTC

Synopsis

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

Arguments

crtc CRTC in question

Description

This functions restores the vblank interrupt state captured with `drm_vblank_off` again. Note that calls to `drm_vblank_on` and `drm_vblank_off` can be unbalanced and so can also be unconditionally called in driver load code to reflect the current hardware state of the crtc.

This is the native kms version of `drm_vblank_on`.

Name

`drm_vblank_pre_modeset` — account for vblanks across mode sets

Synopsis

```
void drm_vblank_pre_modeset (struct drm_device * dev, int crtc);
```

Arguments

dev DRM device

crtc CRTC in question

Description

Account for vblank events across mode setting events, which will likely reset the hardware frame counter.

This is done by grabbing a temporary vblank reference to ensure that the vblank interrupt keeps running across the modeset sequence. With this the software-side vblank frame counting will ensure that there are no jumps or discontinuities.

Unfortunately this approach is racy and also doesn't work when the vblank interrupt stops running, e.g. across system suspend resume. It is therefore highly recommended that drivers use the newer `drm_vblank_off` and `drm_vblank_on` instead. `drm_vblank_pre_modeset` only works correctly when using “cooked” software vblank frame counters and not relying on any hardware counters.

Drivers must call `drm_vblank_post_modeset` when re-enabling the same `crtc` again.

Name

`drm_vblank_post_modeset` — undo `drm_vblank_pre_modeset` changes

Synopsis

```
void drm_vblank_post_modeset (struct drm_device * dev, int crtc);
```

Arguments

dev DRM device

crtc CRTC in question

Description

This function again drops the temporary vblank reference acquired in `drm_vblank_pre_modeset`.

Name

`drm_handle_vblank` — handle a vblank event

Synopsis

```
bool drm_handle_vblank (struct drm_device * dev, int crtc);
```

Arguments

dev DRM device

crtc where this event occurred

Description

Drivers should call this routine in their vblank interrupt handlers to update the vblank counter and send any signals that may be pending.

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

Legacy Support Code

The section very briefly covers some of the old legacy support code which is only used by old DRM drivers which have done a so-called shadow-attach to the underlying device instead of registering as a real driver. This also includes some of the old generic buffer management and command submission code. Do not use any of this in new and modern drivers.

Legacy 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 which *only* work with the legacy shadow-attach driver registration functions. 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`.

Legacy 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

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 `card<num>`. Additionally, a currently unused control node, called `controlID<num>` 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 `DRIVER_RENDER` 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 `renderD<num>`. There will be one render node per device. No ioctls except `PRIME`-related ioctls will be allowed on this node. Especially `GEM_OPEN` 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 `DRM_RENDER_ALLOW` 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 was only used for user-mode-setting drivers around modesetting changes to allow the kernel to update the vblank interrupt after mode setting, since on many devices the vertical blank counter is reset to 0 at some point during modeset. Modern drivers should not call this any more since with kernel mode setting it is a no-op.

Part II. DRM Drivers

This second part of the DRM Developer's Guide documents driver code, implementation details and also all the driver-specific userspace interfaces. Especially since all hardware-acceleration interfaces to userspace are driver specific for efficiency and other reasons these interfaces can be rather substantial. Hence every driver has its own chapter.

Table of Contents

4. drm/i915 Intel GFX Driver	337
Display Hardware Handling	337
Mode Setting Infrastructure	337
Plane Configuration	337
Output Probing	337
DPIO	337
Memory Management and Command Submission	338
Batchbuffer Parsing	338

Chapter 4. drm/i915 Intel GFX Driver

The drm/i915 driver supports all (with the exception of some very early models) integrated GFX chipsets with both Intel display and rendering blocks. This excludes a set of SoC platforms with an SGX rendering unit, those have basic support through the gma500 drm driver.

Display Hardware Handling

This section covers everything related to the display hardware including the mode setting infrastructure, plane, sprite and cursor handling and display, output probing and related topics.

Mode Setting Infrastructure

The i915 driver is thus far the only DRM driver which doesn't use the common DRM helper code to implement mode setting sequences. Thus it has its own tailor-made infrastructure for executing a display configuration change.

Plane Configuration

This section covers plane configuration and composition with the primary plane, sprites, cursors and overlays. This includes the infrastructure to do atomic vsync'ed updates of all this state and also tightly coupled topics like watermark setup and computation, framebuffer compression and panel self refresh.

Output Probing

This section covers output probing and related infrastructure like the hotplug interrupt storm detection and mitigation code. Note that the i915 driver still uses most of the common DRM helper code for output probing, so those sections fully apply.

DPIO

VLV and CHV have slightly peculiar display PHYs for driving DP/HDMI ports. DPIO is the name given to such a display PHY. These PHYs don't follow the standard programming model using direct MMIO registers, and instead their registers must be accessed through IOSF sideband. VLV has one such PHY for driving ports B and C, and CHV adds another PHY for driving port D. Each PHY responds to specific IOSF-SB port.

Each display PHY is made up of one or two channels. Each channel houses a common lane part which contains the PLL and other common logic. CH0 common lane also contains the IOSF-SB logic for the Common Register Interface (CRI) ie. the DPIO registers. CRI clock must be running when any DPIO registers are accessed.

In addition to having their own registers, the PHYs are also controlled through some dedicated signals from the display controller. These include PLL reference clock enable, PLL enable, and CRI clock selection, for example.

Each channel also has two splines (also called data lanes), and each spline is made up of one Physical Access Coding Sub-Layer (PCS) block and two TX lanes. So each channel has two PCS blocks and four TX lanes. The TX lanes are used as DP lanes or TMDS data/clock pairs depending on the output type.

Additionally the PHY also contains an AUX lane with AUX blocks for each channel. This is used for DP AUX communication, but this fact isn't really relevant for the driver since AUX is controlled from the display controller side. No DPIO registers need to be accessed during AUX communication,

Generally the common lane corresponds to the pipe and the spline (PCS/TX) corresponds to the port.

For dual channel PHY (VLV/CHV):

pipe A == CMN/PLL/REF CH0

pipe B == CMN/PLL/REF CH1

port B == PCS/TX CH0

port C == PCS/TX CH1

This is especially important when we cross the streams ie. drive port B with pipe B, or port C with pipe A.

For single channel PHY (CHV):

pipe C == CMN/PLL/REF CH0

port D == PCS/TX CH0

Note: digital port B is DDI0, digital port C is DDI1, digital port D is DDI2

Table 4.1. Dual channel PHY (VLV/CHV)

CH0				CH1			
CMN/PLL/REF				CMN/PLL/REF			
PCS01		PCS23		PCS01		PCS23	
TX0	TX1	TX2	TX3	TX0	TX1	TX2	TX3
DDI0				DDI1			

Table 4.2. Single channel PHY (CHV)

CH0			
CMN/PLL/REF			
PCS01		PCS23	
TX0	TX1	TX2	TX3
DDI2			

Memory Management and Command Submission

This sections covers all things related to the GEM implementation in the i915 driver.

Batchbuffer Parsing

Motivation: Certain OpenGL features (e.g. transform feedback, performance monitoring) require userspace code to submit batches containing commands such as MI_LOAD_REGISTER_IMM to access various registers. Unfortunately, some generations of the hardware will noop these commands in “unsecure” batches (which includes all userspace batches submitted via i915) even though the commands may be safe and represent the intended programming model of the device.

The software command parser is similar in operation to the command parsing done in hardware for unsecure batches. However, the software parser allows some operations that would be noop'd by hardware, if the parser determines the operation is safe, and submits the batch as “secure” to prevent hardware parsing.

Threats: At a high level, the hardware (and software) checks attempt to prevent granting userspace undue privileges. There are three categories of privilege.

First, commands which are explicitly defined as privileged or which should only be used by the kernel driver. The parser generally rejects such commands, though it may allow some from the drm master process.

Second, commands which access registers. To support correct/enhanced userspace functionality, particularly certain OpenGL extensions, the parser provides a whitelist of registers which userspace may safely access (for both normal and drm master processes).

Third, commands which access privileged memory (i.e. GGTT, HWS page, etc). The parser always rejects such commands.

The majority of the problematic commands fall in the MI_* range, with only a few specific commands on each ring (e.g. PIPE_CONTROL and MI_FLUSH_DW).

Implementation: Each ring maintains tables of commands and registers which the parser uses in scanning batch buffers submitted to that ring.

Since the set of commands that the parser must check for is significantly smaller than the number of commands supported, the parser tables contain only those commands required by the parser. This generally works because command opcode ranges have standard command length encodings. So for commands that the parser does not need to check, it can easily skip them. This is implemented via a per-ring length decoding vfunc.

Unfortunately, there are a number of commands that do not follow the standard length encoding for their opcode range, primarily amongst the MI_* commands. To handle this, the parser provides a way to define explicit “skip” entries in the per-ring command tables.

Other command table entries map fairly directly to high level categories mentioned above: rejected, master-only, register whitelist. The parser implements a number of checks, including the privileged memory checks, via a general bitmasking mechanism.

Name

`i915_cmd_parser_init_ring` — set cmd parser related fields for a ringbuffer

Synopsis

```
int i915_cmd_parser_init_ring (struct intel_engine_cs * ring);
```

Arguments

ring the ringbuffer to initialize

Description

Optionally initializes fields related to batch buffer command parsing in the struct `intel_engine_cs` based on whether the platform requires software command parsing.

Return

non-zero if initialization fails

Name

`i915_cmd_parser_fini_ring` — clean up cmd parser related fields

Synopsis

```
void i915_cmd_parser_fini_ring (struct intel_engine_cs * ring);
```

Arguments

ring the ringbuffer to clean up

Description

Releases any resources related to command parsing that may have been initialized for the specified ring.

Name

`i915_needs_cmd_parser` — should a given ring use software command parsing?

Synopsis

```
bool i915_needs_cmd_parser (struct intel_engine_cs * ring);
```

Arguments

ring the ring in question

Description

Only certain platforms require software batch buffer command parsing, and only when enabled via module paramter.

Return

true if the ring requires software command parsing

Name

`i915_parse_cmds` — parse a submitted batch buffer for privilege violations

Synopsis

```
int    i915_parse_cmds    (struct    intel_engine_cs    *    ring,    struct
drm_i915_gem_object    *    batch_obj,    u32    batch_start_offset,    bool
is_master);
```

Arguments

<i>ring</i>	the ring on which the batch is to execute
<i>batch_obj</i>	the batch buffer in question
<i>batch_start_offset</i>	byte offset in the batch at which execution starts
<i>is_master</i>	is the submitting process the drm master?

Description

Parses the specified batch buffer looking for privilege violations as described in the overview.

Return

non-zero if the parser finds violations or otherwise fails

Name

`i915_cmd_parser_get_version` — get the cmd parser version number

Synopsis

```
int i915_cmd_parser_get_version ( void );
```

Arguments

void no arguments

Description

The cmd parser maintains a simple increasing integer version number suitable for passing to userspace clients to determine what operations are permitted.

Return

the current version number of the cmd parser