Linux DRM Developer's Guide

Jesse Barnes

Initial version
Intel Corporation

<jesse.barnes@intel.com>

Laurent Pinchart

Driver internals
Ideas on board SPRL

<laurent.pinchart@ideasonboard.com>

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

Table of Contents

Chapter 1. Introduction

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

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

[Insert diagram of typical DRM stack here]

Chapter 2. DRM Internals

Table of Contents

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

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

The DRM layer provides several services to graphics drivers, many of them driven by the application interfaces it provides through libdrm, the library that wraps most of the DRM ioctls. These include vblank event handling, memory management, output management, framebuffer management, command submission & fencing, suspend/resume support, and DMA services.

Driver Initialization

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

The drm_driver structure contains static information that describes the driver and features it supports, and pointers to methods that the DRM core will call to implement the DRM API. We will first go through the drm_driver static information fields, and will then describe individual operations in details as they get used in later sections.

Driver Information

Driver Features

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

```
u32 driver_features;
```

Driver Feature Flags

DRIVER_USE_AGP

> Driver uses AGP interface, the DRM core will manage AGP resources.

DRIVER_REQUIRE_AGP

Driver needs AGP interface to function. AGP initialization failure will become a fatal error.

DRIVER_USE_MTRR

Driver uses MTRR interface for mapping memory, the DRM core will manage MTRR resources. Deprecated.

DRIVER_PCI_DMA

Driver is capable of PCI DMA, mapping of PCI DMA buffers to userspace will be enabled. Deprecated.

DRIVER_SG

Driver can perform scatter/gather DMA, allocation and mapping of scatter/gather buffers will be enabled. Deprecated.

DRIVER_HAVE_DMA

Driver supports DMA, the userspace DMA API will be supported. Deprecated.

DRIVER_HAVE_IRQ, DRIVER_IRQ_SHARED

DRIVER_HAVE_IRQ indicates whether the driver has an IRQ handler. The DRM core will automatically register an interrupt handler when the flag is set. DRIVER_IRQ_SHARED indicates whether the device & handler support shared IRQs (note that this is required of PCI drivers).

DRIVER_IRQ_VBL

Unused. Deprecated.

DRIVER_DMA_QUEUE

Should be set if the driver queues DMA requests and completes them asynchronously. Deprecated.

DRIVER_FB_DMA

Driver supports DMA to/from the framebuffer, mapping of frambuffer DMA buffers to userspace will be supported. Deprecated.

DRIVER_IRQ_VBL2

Unused. Deprecated.

DRIVER_GEM

Driver use the GEM memory manager.

DRIVER_MODESET

Driver supports mode setting interfaces (KMS).

DRIVER_PRIME

Driver implements DRM PRIME buffer sharing.

Major, Minor and Patchlevel

```
int major;
int minor;
int patchlevel;
```

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

The major and minor numbers are also used to verify the requested driver API version passed to DRM_IOCTL_SET_VERSION. When the driver API changes between minor versions, applications can call DRM_IOCTL_SET_VERSION to select a specific version of the API. If the requested major isn't equal to the driver major, or the requested minor is larger than the driver minor, the DRM_IOCTL_SET_VERSION call will return an error. Otherwise the driver's set_version() method will be called with the requested version.

Name, Description and Date

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

The driver name is printed to the kernel log at initialization time, used for IRQ registration and passed to userspace through DRM_IOCTL_VERSION.

The driver description is a purely informative string passed to userspace through the DRM_IOCTL_VERSION ioctl and otherwise unused by the kernel.

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

Driver Load

The `load` method is the driver and device initialization entry point. The method is responsible for allocating and initializing driver private data, specifying supported performance counters, performing resource allocation and mapping (e.g. acquiring clocks, mapping registers or allocating command buffers), initializing the memory manager (the section called “Memory management”), installing the IRQ handler (the section called “IRQ Registration”), setting up vertical blanking handling (the section called “Vertical Blanking”), mode setting (the section called “Mode Setting”) and initial output configuration (the section called “KMS Initialization and Cleanup”).

Note

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

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

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

Driver Private & Performance Counters

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

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

IRQ Registration

The DRM core tries to facilitate IRQ handler registration and unregistration by providing `drm_irq_install` and `drm_irq_uninstall` functions. Those functions only support a single interrupt per device.

Both functions get the device IRQ by calling `drm_dev_to_irq`. This inline function will call a bus-specific operation to retrieve the IRQ number. For platform devices, `platform_get_irq`(..., 0) is used to retrieve the IRQ number.

`drm_irq_install` starts by calling the `irq_preinstall` driver operation. The operation is optional and must make sure that the interrupt will not get fired by clearing all pending interrupt flags or disabling the interrupt.

The IRQ will then be requested by a call to `request_irq`. If the DRIVER_IRQ_SHARED driver feature flag is set, a shared (IRQF_SHARED) IRQ handler will be requested.

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

Finally the function calls the optional `irq_postinstall` driver operation. The operation usually enables interrupts (excluding the vblank interrupt, which is enabled separately), but drivers may choose to enable/disable interrupts at a different time.

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

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 accomodate 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 capabitilies and is thus limited to UMA devices.

The Translation Table Manager (TTM)

TTM design background and information belongs here.

TTM initialization

Warning

This section is outdated.

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

The ttm_global_reference structure is made up of several fields:

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

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

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

The Graphics Execution Manager (GEM)

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

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

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

On a fundamental level, GEM involves several operations:

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

Buffer object allocation is relatively straightforward and largely provided by Linux's shmem layer, which provides memory to back each object.

Device-specific operations, such as command execution, pinning, buffer read & write, mapping, and domain ownership transfers are left to driver-specific ioctls.

GEM Initialization

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

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

GEM Objects Creation

GEM splits creation of GEM objects and allocation of the memory that backs them in two distinct operations.

GEM objects are represented by an instance of struct drm_gem_object. Drivers usually need to extend GEM objects with private information and thus create a driver-specific GEM object structure type that embeds an instance of struct drm_gem_object.

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

GEM uses shmem to allocate anonymous pageable memory. `drm_gem_object_init` will create an shmfs file of the requested size and store it into the struct drm_gem_object *`filp`* field. The memory is used as either main storage for the object when the graphics hardware uses system memory directly or as a backing store otherwise.

Drivers are responsible for the actual physical pages allocation by calling `shmem_read_mapping_page_gfp` for each page. 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 shmfs backing (called private GEM objects) by initializing them with a call to `drm_gem_private_object_init` instead of `drm_gem_object_init`. Storage for private GEM objects must be managed by drivers.

Drivers that do not need to extend GEM objects with private information can call the `drm_gem_object_alloc` function to allocate and initialize a struct drm_gem_object instance. The GEM core will call the optional driver `gem_init_object` operation after initializing the GEM object with `drm_gem_object_init`.

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

No alloc-and-init function exists for private GEM objects.

GEM Objects Lifetime

All GEM objects are reference-counted by the GEM core. References can be acquired and 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 shmfs backing store must be released by calling `drm_gem_object_release` (that function can safely be called if no shmfs backing store has been created).

GEM Objects Naming

Communication between userspace and the kernel refers to GEM objects using local handles, global names or, more recently, file descriptors. All of those are 32-bit integer values; the usual Linux kernel limits apply to the file descriptors.

GEM handles are local to a DRM file. Applications get a handle to a GEM object through a driver-specific ioctl, and can use that handle to refer to the GEM object in other standard or driver-specific ioctls. Closing a DRM file handle frees all its GEM handles and dereferences the associated GEM objects.

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

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

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

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

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

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

Those two operations convert a handle to a PRIME file descriptor and vice versa. Drivers must use the kernel dma-buf buffer sharing framework to manage the PRIME file descriptors.

While non-GEM drivers must implement the operations themselves, GEM drivers must use the `drm_gem_prime_handle_to_fd` and `drm_gem_prime_fd_to_handle` helper functions. Those helpers rely on the driver `gem_prime_export` and `gem_prime_import` operations to create a dma-buf instance from a GEM object (dma-buf exporter role) and to create a GEM object from a dma-buf instance (dma-buf importer role).

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

These two operations are mandatory for GEM drivers that support DRM PRIME.

GEM Objects Mapping

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

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

The second method uses the mmap system call on the DRM file handle.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
           off_t offset);
```

DRM identifies the GEM object to be mapped by a fake offset passed through the mmap offset argument. Prior to being mapped, a GEM object must thus be associated with a fake offset. To do so, drivers must call `drm_gem_create_mmap_offset` on the object. The function allocates a fake offset range from a pool and stores the offset divided by PAGE_SIZE in `obj->map_list.hash.key`. Care must be taken not to call `drm_gem_create_mmap_offset` if a fake offset has already been allocated for the object. This can be tested by `obj->map_list.map` being non-NULL.

Once allocated, the fake offset value (`obj->map_list.hash.key << PAGE_SHIFT`) must be passed to the application in a driver-specific way and can then be used as the mmap offset argument.

The GEM core provides a helper method `drm_gem_mmap` to handle object mapping. The method can be set directly as the mmap file operation handler. It will look up the GEM object based on the offset value and set the VMA operations to the drm_driver *`gem_vm_ops`* field. Note that `drm_gem_mmap` doesn't map memory to userspace, but relies on the driver-provided fault handler to map pages individually.

To use `drm_gem_mmap`, drivers must fill the struct drm_driver *`gem_vm_ops`* field with a pointer to VM operations.

```
struct vm_operations_struct *gem_vm_ops

  struct vm_operations_struct {
          void (*open)(struct vm_area_struct * area);
          void (*close)(struct vm_area_struct * area);
          int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
  };
```

The `open` and `close` operations must update the GEM object reference count. Drivers can use the `drm_gem_vm_open` and `drm_gem_vm_close` helper functions directly as open and close handlers.

The fault operation handler is responsible for mapping individual pages to userspace when a page fault occurs. Depending on the memory allocation scheme, drivers can allocate pages at fault time, or can decide to allocate memory for the GEM object at the time the object is created.

Drivers that want to map the GEM object upfront instead of handling page faults can implement their own mmap file operation handler.

Dumb GEM Objects

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

Dumb GEM objects partly alleviate the problem by providing a standard API to create dumb buffers suitable for scanout, which can then be used to create KMS frame buffers.

To support dumb GEM objects drivers must implement the `dumb_create`, `dumb_destroy` and `dumb_map_offset` operations.

- `int (*dumb_create)(struct drm_file *file_priv, struct drm_device *dev,`
  `struct drm_mode_create_dumb *args);`

The dumb_create operation creates a GEM object suitable for scanout based on the width, height and depth from the struct drm_mode_create_dumb argument. It fills the argument's *handle*, *pitch* and *size* fields with a handle for the newly created GEM object and its line pitch and size in bytes.

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

  The dumb_destroy operation destroys a dumb GEM object created by dumb_create.

- ```
  int (*dumb_map_offset)(struct drm_file *file_priv, struct drm_device *dev,
                         uint32_t handle, uint64_t *offset);
  ```

  The dumb_map_offset operation associates an mmap fake offset with the GEM object given by the handle and returns it. Drivers must use the drm_gem_create_mmap_offset function to associate the fake offset as described in the section called “GEM Objects Mapping”.

Memory Coherency

When mapped to the device or used in a command buffer, backing pages for an object are flushed to memory and marked write combined so as to be coherent with the GPU. Likewise, if the CPU accesses an object after the GPU has finished rendering to the object, then the object must be made coherent with the CPU's view of memory, usually involving GPU cache flushing of various kinds. This core CPU<->GPU coherency management is provided by a device-specific ioctl, which evaluates an object's current domain and performs any necessary flushing or synchronization to put the object into the desired coherency domain (note that the object may be busy, i.e. an active render target; in that case, setting the domain blocks the client and waits for rendering to complete before performing any necessary flushing operations).

Command Execution

Perhaps the most important GEM function for GPU devices is providing a command execution interface to clients. Client programs construct command buffers containing references to previously allocated memory objects, and then submit them to GEM. At that point, GEM takes care to bind all the objects into the GTT, execute the buffer, and provide necessary synchronization between clients accessing the same buffers. This often involves evicting some objects from the GTT and re-binding others (a fairly expensive operation), and providing relocation support which hides fixed GTT offsets from clients. Clients must take care not to submit command buffers that reference more objects than can fit in the GTT; otherwise, GEM will reject them and no rendering will occur. Similarly, if several objects in the buffer require fence registers to be allocated for correct rendering (e.g. 2D blits on pre-965 chips), care must be taken not to require more fence registers than are available to the client. Such resource management should be abstracted from the client in libdrm.

Mode Setting

Drivers must initialize the mode setting core by calling `drm_mode_config_init` on the DRM device. The function initializes the drm_device *`mode_config`* field and never fails. Once done, mode configuration must be setup by initializing the following fields.

- `int min_width, min_height;`
  `int max_width, max_height;`

  Minimum and maximum width and height of the frame buffers in pixel units.

- `struct drm_mode_config_funcs *funcs;`

  Mode setting functions.

Frame Buffer Creation

```
struct drm_framebuffer *(*fb_create)(struct drm_device *dev,
                                     struct drm_file *file_priv,
                                     struct drm_mode_fb_cmd2 *mode_cmd);
```

Frame buffers are abstract memory objects that provide a source of pixels to scanout to a CRTC. Applications explicitly request the creation of frame buffers through the DRM_IOCTL_MODE_ADDFB(2) ioctls and receive an opaque handle that can be passed to the KMS CRTC control, plane configuration and page flip functions.

Frame buffers rely on the underneath memory manager for low-level memory operations. When creating a frame buffer applications pass a memory handle (or a list of memory handles for multi-planar formats) through the *`drm_mode_fb_cmd2`* argument. This document assumes that the driver uses GEM, those handles thus reference GEM objects.

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

If the parameters are deemed valid, drivers then create, initialize and return an instance of struct drm_framebuffer. If desired the instance can be embedded in a larger driver-specific structure. The new instance is initialized with a call to `drm_framebuffer_init` which takes a pointer to DRM frame buffer operations (struct drm_framebuffer_funcs). 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.

After initializing the drm_framebuffer instance 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.

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.

KMS Initialization and Cleanup

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

CRTCs (struct drm_crtc)

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

CRTC Initialization

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

CRTC Operations

Set Configuration

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

Apply a new CRTC configuration to the device. The configuration specifies a CRTC, a frame buffer to scan out from, a (x,y) position in the frame buffer, a display mode and an array of connectors to drive with the CRTC if possible.

If the frame buffer specified in the configuration is NULL, the driver must detach all encoders connected to the CRTC and all connectors attached to those encoders and disable them.

This operation is called with the mode config lock held.

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

Page Flipping

```
int (*page_flip)(struct drm_crtc *crtc, struct drm_framebuffer *fb,
                 struct drm_pending_vblank_event *event);
```

Schedule a page flip to the given frame buffer for the CRTC. This operation is called with the mode config mutex held.

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

The `page_flip` operation schedules a page flip. Once any pending rendering targetting 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 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 (*gamma_set)(struct drm_crtc *crtc, u16 *r, u16 *g, u16 *b, uint32_t start, uint32_t size);`

  Apply a gamma table to the device. The operation is optional.

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

  Destroy the CRTC when not needed anymore. See the section called “KMS Initialization and Cleanup”.

Planes (struct drm_plane)

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

Plane Initialization

Planes are optional. To create a plane, a KMS 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_plane_init`. The function takes a bitmask of the CRTCs that can be associated with the plane, a pointer to the plane functions and a list of format supported formats.

Plane Operations

- `int (*update_plane)(struct drm_plane *plane, struct drm_crtc *crtc, struct drm_framebuffer *fb, int crtc_x, int crtc_y, unsigned int crtc_w, unsigned int crtc_h, uint32_t src_x, uint32_t src_y, uint32_t src_w, uint32_t src_h);`

  Enable and configure the plane to use the given CRTC and frame buffer.

  The source rectangle in frame buffer memory coordinates is given by the *`src_x`*, *`src_y`*, *`src_w`* and *`src_h`* parameters (as 16.16 fixed point values). Devices that don't support subpixel plane coordinates can ignore the fractional part.

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

- `int (*disable_plane)(struct drm_plane *plane);`

  Disable the plane. The DRM core calls this method in response to a DRM_IOCTL_MODE_SETPLANE ioctl call with the frame buffer ID set to 0. Disabled planes must not be processed by the CRTC.

- `void (*destroy)(struct drm_plane *plane);`

  Destroy the plane when not needed anymore. See the section called “KMS Initialization and Cleanup”.

Encoders (struct drm_encoder)

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

Encoder Initialization

As for CRTCs, a KMS driver must create, initialize and register at least one struct drm_encoder instance. The instance is allocated and zeroed by the driver, possibly as part of a larger structure.

Drivers must initialize the struct drm_encoder *possible_crtcs* and *possible_clones* fields before registering the encoder. Both fields are bitmasks of respectively the CRTCs that the encoder can be connected to, and sibling encoders candidate for cloning.

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

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

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

Encoder Operations

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

  Called to destroy the encoder when not needed anymore. See the section called “KMS Initialization and Cleanup”.

Connectors (struct drm_connector)

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

Connector Initialization

Finally a KMS driver must create, initialize, register and attach at least one struct drm_connector instance. The instance is created as other KMS objects and initialized by setting the following fields.

`interlace_allowed`

Whether the connector can handle interlaced modes.

`doublescan_allowed`

Whether the connector can handle doublescan.

`display_info`

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

`polled`

Connector polling mode, a combination of

DRM_CONNECTOR_POLL_HPD

The connector generates hotplug events and doesn't need to be periodically polled. The CONNECT and DISCONNECT flags must not be set together with the HPD flag.

DRM_CONNECTOR_POLL_CONNECT

Periodically poll the connector for connection.

DRM_CONNECTOR_POLL_DISCONNECT

Periodically poll the connector for disconnection.

Set to 0 for connectors that don't support connection status discovery.

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

Supported connector types are

- DRM_MODE_CONNECTOR_VGA
- DRM_MODE_CONNECTOR_DVII
- DRM_MODE_CONNECTOR_DVID
- DRM_MODE_CONNECTOR_DVIA
- DRM_MODE_CONNECTOR_Composite
- DRM_MODE_CONNECTOR_SVIDEO
- DRM_MODE_CONNECTOR_LVDS
- DRM_MODE_CONNECTOR_Component
- DRM_MODE_CONNECTOR_9PinDIN
- DRM_MODE_CONNECTOR_DisplayPort
- DRM_MODE_CONNECTOR_HDMIA
- DRM_MODE_CONNECTOR_HDMIB
- DRM_MODE_CONNECTOR_TV
- DRM_MODE_CONNECTOR_eDP
- DRM_MODE_CONNECTOR_VIRTUAL

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

Finally, drivers must initialize the connectors state change detection with a call to `drm_kms_helper_poll_init`. If at least one connector is pollable but can't generate hotplug interrupts (indicated by the DRM_CONNECTOR_POLL_CONNECT and DRM_CONNECTOR_POLL_DISCONNECT connector flags), a delayed work will automatically be queued to periodically poll for changes. Connectors that can generate hotplug interrupts must be marked with the DRM_CONNECTOR_POLL_HPD flag instead, and their interrupt handler must call `drm_helper_hpd_irq_event`. The function will queue a delayed work to check the state of all connectors, but no periodic polling will be done.

Connector Operations

> **Note**
>
> Unless otherwise state, all operations are mandatory.

DPMS

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

The DPMS operation sets the power state of a connector. The mode argument is one of

- DRM_MODE_DPMS_ON
- DRM_MODE_DPMS_STANDBY
- DRM_MODE_DPMS_SUSPEND
- DRM_MODE_DPMS_OFF

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

Modes

```
int (*fill_modes)(struct drm_connector *connector, uint32_t max_width,
                  uint32_t max_height);
```

Fill the mode list with all supported modes for the connector. If the *max_width* and *max_height* arguments are non-zero, the implementation must ignore all modes wider than *max_width* or higher than *max_height*.

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

Connection Status

The connection status is updated through polling or hotplug events when supported (see *polled*). The status value is reported to userspace through ioctls and must not be used inside the driver, as it only gets initialized by a call to `drm_mode_getconnector` from userspace.

```
enum drm_connector_status (*detect)(struct drm_connector *connector,
                                    bool force);
```

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

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

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

Miscellaneous

- `void (*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.

Mode Setting Helper Functions

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

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

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

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

Helper Functions

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

  The `drm_crtc_helper_set_config` helper function is a CRTC `set_config` implementation. It first tries to locate the best encoder for each connector by calling the connector `best_encoder` helper operation.

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

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

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

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

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

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

  The `drm_helper_probe_single_connector_modes` helper function is a connector `fill_modes` implementation that updates the connection status for the connector and then retrieves a list of modes by calling the connector `get_modes` helper operation.

  The function filters out modes larger than *`max_width`* and *`max_height`* if specified. It then calls the connector `mode_valid` helper operation for each mode in the probed list to check whether the mode is valid for the connector.

CRTC Helper Operations

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

  Let CRTCs 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);`

  ### Note

  FIXME: The mode argument be const, but the i915 driver modifies mode->clock in `intel_dp_mode_fixup`.

  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 connecter. Device that map connectors to encoders 1:1 simply return the pointer to the associated encoder. This operation is mandatory.

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

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

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

- `__u32 type;`

  Mode type bitmask, a combination of

  DRM_MODE_TYPE_BUILTIN

  not used?

  DRM_MODE_TYPE_CLOCK_C

  not used?

  DRM_MODE_TYPE_CRTC_C

  not used?

  DRM_MODE_TYPE_PREFERRED - The preferred mode for the connector

  not used?

  DRM_MODE_TYPE_DEFAULT

  not used?

  DRM_MODE_TYPE_USERDEF

  not used?

  DRM_MODE_TYPE_DRIVER

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

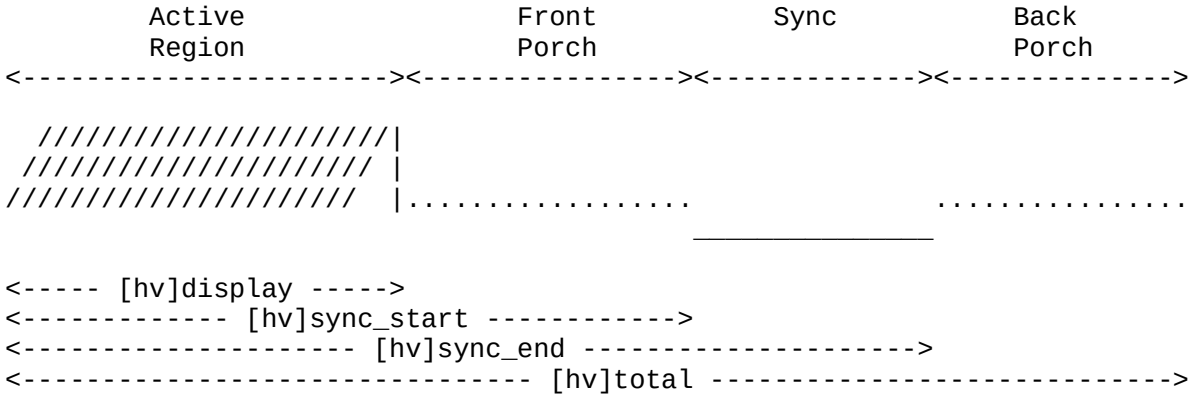
Drivers must set the DRM_MODE_TYPE_DRIVER bit for all modes they create, and set the DRM_MODE_TYPE_PREFERRED bit for the preferred mode.

- `__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 initilization time when a fixed-size panel is attached to the connector). The mode *`width_mm`* and *`height_mm`* fields are only used internally during EDID parsing and should not be set when creating modes manually.

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

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

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

Note

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

Modeset Helper Functions Reference

Name

drm_helper_move_panel_connectors_to_head — move panels to the front in the connector list

Synopsis

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

Arguments

dev

drm device to operate on

Description

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

Name

drm_helper_probe_single_connector_modes — get complete set of display modes

Synopsis

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

Arguments

connector

connector to probe

maxX

max width for modes

maxY

max height for modes

LOCKING

Caller must hold mode config lock.

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

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

RETURNS

Number of modes found on *connector*.

Name

drm_helper_encoder_in_use — check if a given encoder is in use

Synopsis

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

Arguments

encoder

encoder to check

LOCKING

Caller must hold mode config lock.

Walk *encoders*'s DRM device's mode_config and see if it's in use.

RETURNS

True if *encoder* is part of the mode_config, false otherwise.

Name

drm_helper_crtc_in_use — check if a given CRTC is in a mode_config

Synopsis

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

Arguments

crtc

CRTC to check

LOCKING

Caller must hold mode config lock.

Walk *crtc*'s DRM device's mode_config and see if it's in use.

RETURNS

True if *crtc* is part of the mode_config, false otherwise.

Name

drm_helper_disable_unused_functions — disable unused objects

Synopsis

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

Arguments

dev

DRM device

LOCKING

Caller must hold mode config lock.

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

Name

drm_crtc_helper_set_mode — internal helper to set a mode

Synopsis

```
bool fsfuncdrm_crtc_helper_set_mode (crtc,
                                     mode,
                                     x,
```

y,
old_fb);

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

Arguments

crtc

CRTC to program

mode

mode to use

x

horizontal offset into the surface

y

vertical offset into the surface

old_fb

old framebuffer, for cleanup

LOCKING

Caller must hold mode config lock.

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

RETURNS

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

Name

drm_crtc_helper_set_config — set a new config from userspace

Synopsis

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

Arguments

set

mode set configuration

LOCKING

Caller must hold mode config lock.

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

RETURNS

Returns 0 on success, -ERRNO on failure.

Name

drm_helper_connector_dpms — connector dpms helper implementation

Synopsis

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

Arguments

connector

affected connector

mode

DPMS mode

Description

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

fbdev Helper Functions Reference

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

Name

drm_fb_helper_restore — restore the framebuffer console (kernel) config

Synopsis

```
void fsfuncdrm_fb_helper_restore (void);
void;
```

Arguments

void

no arguments

fbdev helpers

Restore's the kernel's fbcon mode, used for lastclose & panic paths.

Name

drm_fb_helper_initial_config — setup a sane initial connector configuration

Synopsis

```
bool fsfuncdrm_fb_helper_initial_config (fb_helper,
                                         bpp_sel);
struct drm_fb_helper * fb_helper;
int bpp_sel;
```

Arguments

fb_helper

fb_helper device struct

bpp_sel

bpp value to use for the framebuffer configuration

LOCKING

Called at init time by the driver to set up the *fb_helper* initial configuration, must take the mode config lock.

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

RETURNS

Zero if everything went ok, nonzero otherwise.

Name

drm_fb_helper_hotplug_event — respond to a hotplug notification by probing all the outputs attached to the fb

Synopsis

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

Arguments

fb_helper

the drm_fb_helper

LOCKING

Called at runtime, must take mode config lock.

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

RETURNS

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

Display Port Helper Functions Reference

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

Name

struct i2c_algo_dp_aux_data — driver interface structure for i2c over dp aux algorithm

Synopsis

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

Members

running

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

address

i2c target address for the currently ongoing transfer

aux_ch

driver callback to transfer a single byte of the i2c payload

Name

i2c_dp_aux_add_bus — register an i2c adapter using the aux ch helper

Synopsis

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

Arguments

adapter

i2c adapter to register

Description

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

RETURNS

0 on success, -ERRNO on failure.

Vertical Blanking

Vertical blanking plays a major role in graphics rendering. To achieve tear-free display, users must synchronize page flips and/or rendering to vertical blanking. The DRM API offers ioctls to perform page flips synchronized to vertical blanking and wait for vertical blanking.

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

- ```
  int (*enable_vblank) (struct drm_device *dev, int crtc);
  void (*disable_vblank) (struct drm_device *dev, int crtc);
  ```

  Enable or disable vertical blanking interrupts for the given CRTC.

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

  Retrieve the value of the vertical blanking counter for the given CRTC. If the hardware maintains a vertical blanking counter its value should be returned. Otherwise drivers can use the `drm_vblank_count` helper function to handle this operation.

Drivers must initialize the vertical blanking handling core with a call to `drm_vblank_init` in their `load` operation. The function will set the struct drm_device *`vblank_disable_allowed`* field to 0. This will keep vertical blanking interrupts enabled permanently until the first mode set operation, where *`vblank_disable_allowed`* is set to 1. The reason behind this is not clear. Drivers can set the field to 1 after `calling drm_vblank_init` to make vertical blanking interrupts dynamically managed from the beginning.

Vertical blanking interrupts can be enabled by the DRM core or by drivers themselves (for instance to handle page flipping operations). The DRM core maintains a vertical blanking use count to ensure that the interrupts are not disabled while a user still needs them. To increment the use count, drivers call `drm_vblank_get`. Upon return vertical blanking interrupts are guaranteed to be enabled.

To decrement the use count drivers call `drm_vblank_put`. Only when the use count drops to zero will the DRM core disable the vertical blanking interrupts after a delay by scheduling a timer. The delay is accessible through the vblankoffdelay module parameter or the `drm_vblank_offdelay` global variable and expressed in milliseconds. Its default value is 5000 ms.

When a vertical blanking interrupt occurs drivers only need to call the `drm_handle_vblank` function to account for the interrupt.

Resources allocated by `drm_vblank_init` must be freed with a call to `drm_vblank_cleanup` in the driver `unload` operation handler.

Open/Close, File Operations and IOCTLs

Open and Close

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

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

The `firstopen` method is called by the DRM core when an application opens a device that has no other opened file handle. Similarly the `lastclose` method is called when the last application holding a file handle opened on the device closes it. Both methods are mostly used for UMS (User Mode Setting) drivers to acquire and release device resources which should be done in the `load` and `unload` methods for KMS drivers.

Note that the `lastclose` 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.

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,
```

```
        .fasync = drm_fasync,
        .llseek = no_llseek,
```

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

```
        .mmap = drm_gem_mmap,
```

No other file operation is supported by the DRM API.

IOCTLs

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

Driver-specific ioctls descriptors table.

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

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

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

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

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

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

- DRM_AUTH - Only authenticated callers allowed
- DRM_MASTER - The ioctl can only be called on the master file handle

- DRM_ROOT_ONLY - Only callers with the SYSADMIN capability allowed
- DRM_CONTROL_ALLOW - The ioctl can only be called on a control device
- DRM_UNLOCKED - The ioctl handler will be called without locking the DRM global mutex

Command submission & fencing

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

Suspend/Resume

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

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

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

DMA services

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

Chapter 3. Userland interfaces

Table of Contents

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.

VBlank event handling

The DRM core exposes two vertical blank related ioctls:

DRM_IOCTL_WAIT_VBLANK

This takes a struct drm_wait_vblank structure as its argument, and it is used to block or request a signal when a specified vblank event occurs.

DRM_IOCTL_MODESET_CTL

This should be called by application level drivers before and after mode setting, since on many devices the vertical blank counter is reset at that time. Internally, the DRM snapshots the last vblank count when the ioctl is called with the _DRM_PRE_MODESET command, so that the counter won't go backwards (which is dealt with when _DRM_POST_MODESET is used).

Appendix A. DRM Driver API

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