Linux DRM Developer’s Guide

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 can 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'll go over some typical driver initialization requirements, like setting up command buffers, creating an initial output configuration, and initializing core services. Subsequent sections will 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.

The core of every DRM driver is struct drm_device. Drivers will typically statically initialize a drm_device structure, then pass it to drm_init() at load time.

2.1. Driver initialization

Before calling the DRM initialization routines, the driver must first create and fill out a struct drm_device structure.

```
      static struct drm_driver driver = {
/* don't use mtrr's here, the Xserver or user space app should
 * deal with them for intel hardware.
 */
.driver_features =
    DRIVER_USE_AGP | DRIVER_REQUIRE_AGP |
    DRIVER_HAVE_IRQ | DRIVER_IRQ_SHARED | DRIVER_MODESET,
.load = i915_driver_load,
.unload = i915_driver_unload,
.firstopen = i915_driver_firstopen,
.lastclose = i915_driver_lastclose,
.preclose = i915_driver_preclose,
.save = i915_save,
.restore = i915_restore,
.device_is_agp = i915_driver_device_is_agp,
.get_vblank_counter = i915_get_vblank_counter,
.enable_vblank = i915_enable_vblank,
.disable_vblank = i915_disable_vblank,
.irq_preinstall = i915_driver_irq_preinstall,
.irq_postinstall = i915_driver_irq_postinstall,
.irq_uninstall = i915_driver_irq_uninstall,
.irq_handler = i915_driver_irq_handler,
.reclaim_buffers = drm_core_reclaim_buffers,
 .get_map_ofs = drm_core_get_map_ofs,
 .get_reg_ofs = drm_core_get_reg_ofs,
 .fb_probe = intelfb_probe,
 .fb_remove = intelfb_remove,
 .fb_resize = intelfb_resize,
 .master_create = i915_master_create,
 .master_destroy = i915_master_destroy,
#if defined(CONFIG_DEBUG_FS)
 .debugfs_init = i915_debugfs_init,
 .debugfs_cleanup = i915_debugfs_cleanup,
#endif
 .gem_init_object = i915_gem_init_object,
 .gem_free_object = i915_gem_free_object,
 .gem_vm_ops = &i915_gem_vm_ops,
 .ioctls = i915_ioctls,
 .fops = {
  .owner = THIS_MODULE,
  .open = drm_open,
  .release = drm_release,
  .ioctl = drm_ioctl,
  .mmap = drm_mmap,
  .poll = drm_poll,
  .fasync = drm_fasync,
#ifdef CONFIG_COMPAT
  .compat_ioctl = i915_compat_ioctl,
#endif
  },
 .pci_driver = {
  .name = DRIVER_NAME,
  .id_table = pciidlist,
  .probe = probe,
  .remove = __devexit_p(drm_cleanup_pci),
  },
 .name = DRIVER_NAME,
 .desc = DRIVER_DESC,
 .date = DRIVER_DATE,
 .major = DRIVER_MAJOR,
 .minor = DRIVER_MINOR,
 .patchlevel = DRIVER_PATCHLEVEL,
      };
```

In the example above, taken from the i915 DRM driver, the driver sets several flags indicating what core features it supports. We'll go over the individual callbacks in later sections. Since flags indicate which features your driver supports to the DRM core, you need to set most of them prior to calling drm_init(). Some, like DRIVER_MODESET can be set later based on user supplied parameters, but that's the exception rather than the rule.

Driver flags

DRIVER_USE_AGP

Driver uses AGP interface

DRIVER_REQUIRE_AGP

Driver needs AGP interface to function.

DRIVER_USE_MTRR

Driver uses MTRR interface for mapping memory. Deprecated.

DRIVER_PCI_DMA

Driver is capable of PCI DMA. Deprecated.

DRIVER_SG

Driver can perform scatter/gather DMA. Deprecated.

DRIVER_HAVE_DMA

Driver supports DMA. Deprecated.

DRIVER_HAVE_IRQ
DRIVER_IRQ_SHARED

DRIVER_HAVE_IRQ indicates whether the driver has a IRQ handler, DRIVER_IRQ_SHARED indicates whether the device & handler support shared IRQs (note that this is required of PCI drivers).

DRIVER_DMA_QUEUE

If the driver queues DMA requests and completes them asynchronously, this flag should be set. Deprecated.

DRIVER_FB_DMA

Driver supports DMA to/from the framebuffer. Deprecated.

DRIVER_MODESET

Driver supports mode setting interfaces.

In this specific case, the driver requires AGP and supports IRQs. DMA, as we'll see, is handled by device specific ioctls in this case. It also supports the kernel mode setting APIs, though unlike in the actual i915 driver source, this example unconditionally exports KMS capability.

2.2. Driver load

In the previous section, we saw what a typical drm_driver structure might look like. One of the more important fields in the structure is the hook for the load function.

```
static struct drm_driver driver = {
 ...
 .load = i915_driver_load,
  ...
};
```

The load function has many responsibilities: allocating a driver private structure, specifying supported performance counters, configuring the device (e.g. mapping registers & command buffers), initializing the memory manager, and setting up the initial output configuration.

Note that the tasks performed at driver load time must not conflict with DRM client requirements. 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 pre-memory management aware user level drivers 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. The i915 driver supports the "modeset" module parameter to control whether advanced features are enabled at load time or in legacy fashion. If compatibility is a concern (e.g. with drivers converted over to the new interfaces from the old ones), care must be taken to prevent incompatible device initialization and control with the currently active userspace drivers.

2.2.1. 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 can simply allocate one and set drm_device.dev_priv appropriately; at unload the driver can free it and set drm_device.dev_priv to NULL.

The DRM supports several counters which can be used for rough performance characterization. Note that the DRM stat counter system is not often used by applications, and supporting additional counters is completely optional.

These interfaces are 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 to performance monitoring tools and applications.

2.2.2. Configuring the device

Obviously, device configuration will be device specific. However, there are several common operations: finding a device's PCI resources, mapping them, and potentially setting up an IRQ handler.

Finding & mapping resources is fairly straightforward. The DRM wrapper functions, drm_get_resource_start() and drm_get_resource_len() can be used to find BARs on the given drm_device struct. Once those values have been retrieved, the driver load function can call drm_addmap() to create a new mapping for the BAR in question. Note you'll probably want a drm_local_map_t in your driver private structure to track any mappings you create.

if compatibility with other operating systems isn't a concern (DRM drivers can run under various BSD variants and OpenSolaris), native Linux calls can be used for the above, e.g. pci_resource_* and iomap*/iounmap. See the Linux device driver book for more info.

Once you have a register map, you can use the DRM_READn() and DRM_WRITEn() macros to access the registers on your device, or use driver specific versions to offset into your MMIO space relative to a driver specific base pointer (see I915_READ for example).

If your device supports interrupt generation, you may want to setup an interrupt handler at driver load time as well. This is done using the drm_irq_install() function. If your device supports vertical blank interrupts, it should call drm_vblank_init() to initialize the core vblank handling code before enabling interrupts on your device. This ensures the vblank related structures are allocated and allows the core to handle vblank events.

Once your interrupt handler is registered (it'll use your drm_driver.irq_handler as the actual interrupt handling function), you can safely enable interrupts on your device, assuming any other state your interrupt handler uses is also initialized.

Another task that may be necessary 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 once you've mapped the ROM and extracted any necessary information, be sure to unmap it; on many devices the ROM address decoder is shared with other BARs, so leaving it mapped can cause undesired behavior like hangs or memory corruption.

2.2.3. Memory manager initialization

In order to allocate command buffers, cursor memory, scanout buffers, etc., as well as support the latest features provided by packages like Mesa and the X.Org X server, your driver should support a memory manager.

If your driver supports memory management (it should!), you'll need to set that up at load time as well. How you intialize it depends on which memory manager you're using, TTM or GEM.

2.2.3.1. TTM initialization

TTM (for Translation Table Manager) manages video memory and aperture space for graphics devices. TTM supports both UMA devices and devices with dedicated video RAM (VRAM), i.e. most discrete graphics devices. If your device has dedicated RAM, supporting TTM is desireable. TTM also integrates tightly with your driver specific buffer execution function. See the radeon driver for examples.

The core TTM structure is the ttm_bo_driver struct. It 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 will probably eventually call ttm_mem_global_init and ttm_mem_global_release respectively.

Once your global TTM accounting structure is set up and initialized (done by calling ttm_global_item_ref on the global object you just created), you'll 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 can 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 initalization function.

2.2.3.2. GEM initialization

GEM is an alternative to TTM, designed specifically for UMA devices. It has simpler initialization and execution requirements than TTM, but has no VRAM management capability. Core GEM initialization is comprised of a basic drm_mm_init call to create a GTT DRM MM object, which provides an address space pool for object allocation. In a KMS configuration, the driver will need to allocate and initialize a command ring buffer following basic GEM initialization. Most UMA devices have a so-called "stolen" memory region, which provides space for the initial framebuffer and large, contiguous memory regions required by the device. This space is not typically managed by GEM, and must be initialized separately into its own DRM MM object.

Initialization will be driver specific, and will depend on the architecture of the device. In the case of Intel integrated graphics chips like 965GM, GEM initialization can be done by calling the internal GEM init function, i915_gem_do_init(). Since the 965GM is a UMA device (i.e. it doesn't have dedicated VRAM), GEM will manage making regular RAM available for GPU operations. Memory set aside by the BIOS (called "stolen" memory by the i915 driver) will be managed by the DRM memrange allocator; the rest of the aperture will be managed by GEM.

```
/* Basic memrange allocator for stolen space (aka vram) */
drm_memrange_init(&dev_priv->vram, 0, prealloc_size);
/* Let GEM Manage from end of prealloc space to end of aperture */
i915_gem_do_init(dev, prealloc_size, agp_size);
```

Once the memory manager has been set up, we can allocate the command buffer. In the i915 case, this is also done with a GEM function, i915_gem_init_ringbuffer().

2.2.4. Output configuration

The final initialization task is output configuration. This involves finding and initializing the CRTCs, encoders and connectors for your device, creating an initial configuration and registering a framebuffer console driver.

2.2.4.1. Output discovery and initialization

Several core functions exist to create CRTCs, encoders and connectors, namely drm_crtc_init(), drm_connector_init() and drm_encoder_init(), along with several "helper" functions to perform common tasks.

Connectors should be registered with sysfs once they've been detected and initialized, using the drm_sysfs_connector_add() function. Likewise, when they're removed from the system, they should be destroyed with drm_sysfs_connector_remove().

```
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 (again, taken from the i915 driver), a CRT 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 regsitered with sysfs, to make its properties available to applications.

2.2.4.1.1. Helper functions and core functions

Since many PC-class graphics devices have similar display output designs, the DRM provides a set of helper functions to make output management easier. The core helper routines handle encoder re-routing and disabling of unused functions following mode set. Using the helpers is optional, but recommended for devices with PC-style architectures (i.e. a set of display planes for feeding pixels to encoders which are in turn routed to connectors). Devices with more complex requirements needing finer grained management can opt to use the core callbacks directly.

[Insert typical diagram here.] [Insert OMAP style config here.]

For each encoder, CRTC and connector, several functions must be provided, depending on the object type. Encoder objects need should provide a DPMS (basically on/off) function, mode fixup (for converting requested modes into native hardware timings), and prepare, set and commit functions for use by the core DRM helper functions. Connector helpers need to provide mode fetch and validity functions as well as an encoder matching function for returing an ideal encoder for a given connector. The core connector functions include a DPMS callback, (deprecated) save/restore routines, detection, mode probing, property handling, and cleanup functions.

2.3. VBlank event handling

The DRM core exposes two vertical blank related ioctls: DRM_IOCTL_WAIT_VBLANK and DRM_IOCTL_MODESET_CTL.

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

DRM_IOCTL_MODESET_CTL should be called by application level drivers before and after mode setting, since on many devices the vertical blank counter will be 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).

To support the functions above, the DRM core provides several helper functions for tracking vertical blank counters, and requires drivers to provide several callbacks: get_vblank_counter(), enable_vblank() and disable_vblank(). The core uses get_vblank_counter() to keep the counter accurate across interrupt disable periods. It should return the current vertical blank event count, which is often tracked in a device register. The enable and disable vblank callbacks should enable and disable vertical blank interrupts, respectively. In the absence of DRM clients waiting on vblank events, the core DRM code will use the disable_vblank() function to disable interrupts, which saves power. They'll be re-enabled again when a client calls the vblank wait ioctl above.

Devices that don't provide a count register can simply use an internal atomic counter incremented on every vertical blank interrupt, and can make their enable and disable vblank functions into no-ops.

2.4. Memory management

The memory manager lies at the heart of many DRM operations, and is also required to support advanced client features like OpenGL pbuffers. The DRM currently contains two memory managers, TTM and GEM.

2.4.1. The Translation Table Manager (TTM)

TTM was developed by Tungsten Graphics, primarily by Thomas HellstrÃ¶m, and is intended to be a flexible, high performance graphics memory manager.

Drivers wishing to support TTM must fill out a drm_bo_driver structure.

TTM design background and information belongs here.

2.4.2. The Graphics Execution Manager (GEM)

GEM is an Intel project, authored by Eric Anholt and Keith Packard. It provides simpler interfaces than TTM, and is well suited for UMA devices.

GEM-enabled drivers must provide gem_init_object() and gem_free_object() callbacks to support the core memory allocation routines. They should also provide several driver specific ioctls to support command execution, pinning, buffer read & write, mapping, and domain ownership transfers.

On a fundamental level, GEM involves several operations: memory allocation and freeing, command execution, and 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. When mapped into the GTT or used in a command buffer, the backing pages for an object are flushed to memory and marked write combined so as to be coherent with the GPU. Likewise, when the GPU finishes rendering to an object, if the CPU accesses it, it 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 the GEM set domain function, 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 the set domain function will block the client and wait for rendering to complete before performing any necessary flushing operations).

Perhaps the most important GEM function is providing a command execution interface to clients. Client programs construct command buffers containing references to previously allocated memory objects and submit them to GEM. At that point, GEM will take 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 or 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.

2.5. Output management

At the core of the DRM output management code is a set of structures representing CRTCs, encoders and connectors.

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, 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.

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.

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.

2.6. Framebuffer management

In order to set a mode on a given CRTC, encoder and connector configuration, clients need to provide a framebuffer object which will provide a source of pixels for the CRTC to deliver to the encoder(s) and ultimately the connector(s) in the configuration. A framebuffer is fundamentally a driver specific memory object, made into an opaque handle by the DRM addfb function. Once an fb has been created this way it can be passed to the KMS mode setting routines for use in a configuration.

2.7. Command submission & fencing

This should cover a few device specific command submission implementations.

2.8. Suspend/resume

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

2.9. 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. Only need high level info, since man pages will cover the rest.

Appendix A. DRM Driver API

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