

The Mine of Information

(Nuggets of Programming and Linux)



About

Welcome

Recent Posts

Hyperconvergence

OSNews Various

Learning Python

Thoughts on the Equifax Data Breach

Data Warehousing

The Bitwarden Password Manager

Yubikey, FIDO2 and Backups

More databases - MemSQL and RocksDB

Categories

BigData

Cloud

Cryptography

Git

Infrastructure

Java

Links

Linux

Management

OSGi

Off-topic

OpenWRT

Programming

Security

Site

DRM and KMS kernel module implementations

First published on: October 16, 2012

Categories: [Linux](#)

Intro

The “Direct Rendering Manager” (DRM) and “Kernel Modesetting” (KMS) APIs are important parts of the Linux graphics system. However documentation on exactly what they are is very hard to find - and most of what Google turns up is completely obsolete. It appears that the people *working* in this area are too busy to document it.

An article on this site gives an [overview of the linux graphics stack](#); **this** article presents more detailed information about the internal implementation details of the DRM kernel module.

You may wish to also read [my article on graphics cards](#) to understand roughly how modern graphics cards work.

Update: I have found an article [DRM Developers Guide](#) from 2012 which has very detailed discussion of internal DRM details, and is far more thorough than this article!

The code discussion in this article is based on the “radeon.ko” driver for ATI Radeon graphics cards attached via PCIe (as that is what is present in my laptop). However the general principles should apply to many cards, and at least partially to embedded graphics devices as well as PCIe ones.

Note that I’m no expert in this area, and the existing documentation is awful. There *will be* mistakes and misunderstandings below - possibly significant ones; corrections are welcome. This is also an article-in-progress, and hopefully will improve over time.

Purpose of DRI, DRM and KMS

In the beginning, all graphics was done through the X server ¹; a different “DDX” userspace driver exists for each card which maps shared buffers of the video card by making MMAP system calls against the /dev/mem “file”, then reading/writing addresses in the mapped range. The drivers are responsible for both supporting the X11 drawing APIs (traditional 2D API or X Render extension), as well as setting the graphics mode and generally configuring the graphics card - all without any kernel-side support.

The problem with this approach is that this:

- 1. relies on X having *exclusive* access to the graphics card;
- 2. is not very efficient (kernel-mode code can handle memory mappings etc. much more efficiently);
- 3. puts lots of critical logic in the X-specific drivers where other graphics systems can’t use it.

Item (1) is a major issue - forcing applications wanting to use the 3D rendering capabilities of modern cards to pass all data through the X server is just crazy. The X Render extension allows client apps to at least pass tessellated 2D shapes and character glyphs, but that doesn’t help a lot. There is also a mechanism for passing OpenGL commands to X, and have X run them on behalf of the caller (GLX Indirect Rendering) - but that is also rather inefficient. In the end, it is best for performance-critical apps to just have direct access to the graphics card. However there then needs to be a mechanism for coordinating access from multiple applications.

Even in older systems, assumption (1) is not entirely true: virtual consoles based on framebuffer drivers also need access to the graphics card when `alt - f(n)` is pressed. Without KMS, there are some ugly hacks in X to support this which still don’t work entirely effectively, meaning switching to a console is slow and causes screen flicker. Switching from the framebuffer to X during booting is another case.

Item (2) is also important. Graphics cards use DMA extensively, but driving this from userspace isn’t efficient (in particular, handling interrupts). Doing modern graphics also requires allocating/releasing lots of memory buffers - something that is far more easily done in the kernel.

Item (3) blocks research into and implementation of alternatives to X. While X is a proven system, it is many decades old; it would be nice to allow alternatives to at least be explored without having to port and maintain a fork of many DDX (Device Dependent X) graphics drivers.

The DRM (Direct Rendering Manager) kernel module was developed to deal with the above issues.

LibDRM

The DRM kernel module exposes an “unstable” api to userspace, and should not be directly used by applications. Instead, a C library is available from freedesktop.org (and bundled with all distributions), which exposes a stable api. This library is then used from X, Mesa (opengl) drivers, libva, etc. When features are added to drm drivers (eg new ioctls), a corresponding release of libdrm is made.

This library supports both Linux and BSD, and potentially other operating systems. It contains both generic code, and card-specific code. It then compiles to a set of dynamically-loadable libraries, one for each supported card type (kernel driver), which can usually be found under /usr/lib. For example, on my system the libdrm version for AMD radeon cards can be found at `/usr/lib/lib386-linux-gnu/libdrm_radeon.so.1`.

Reading the libdrm source code - and in particular, headers `xf86drm.h`, `xf86drmMode.h` and `include/drm/libdrm.h` - is probably the best way to start understanding the DRM and KMS kernel implementations.

Direct Rendering Manager (DRM)

The “drm core” module is generic code that applies to all cards; it provides a set of IOCTL operations which it either handles itself or delegates to a card-specific kernel module.

When loaded, a card-specific drm helper module calls into the drm module to register itself as a “drm driver”, and provides a set of function-pointers that the drm core module may invoke. The “drm core” then creates a file `/dev/dri/card{n}` on which IOCTLs can be made to talk to the driver; reading the file also returns “event” structures indicating when things such as “vblank” or “flip complete” events have occurred.

As part of the registration process, the “drm helper module” provides a set of flags indicating which features it supports, for example `DRIVER_MODESET` indicates that it supports kernel modesetting (KMS). The drm core will return an error when userspace attempts to invoke functionality that the card-specific “helper module” does not support.

The drm “api” (ie the api exposed by libdrm) defines the minimum number of extensions necessary to support sharing of a graphics card between multiple userspace applications. An X DDX driver with card-specific knowledge is still needed to access the full card functionality. For example, it is the DDX driver which knows what address-ranges on the card are safe for userspace to mmap; the driver passes this info to the drm module, so later mapping requests from X client applications can be validated. This reduces the amount of complexity needed in the card-specific drivers (which was particularly useful when migrating from completely userspace X drivers to DRM).

Because “drm” (ie the core code) is itself a kernel module, there are corresponding sysfs entries. You will find them at `/sys/virtual/devices/drm`, `/sys/class/drm` and `/sys/module/drm`. DRM drivers declare a dependency on the drm module.

Because the kernel API for DRM is not exposed directly to users (instead being wrapped in libdrm), there is unfortunately little documentation for the IOCTLs exposed by the drm module itself; it is necessary to read the source-code for libdrm to understand how these ioctls are actually used. As noted above, understanding the libdrm code is really a prerequisite to understanding the drm kernel side.

KMS

The first version of DRM only provided support for “DMA style buffer transfer” between memory that the userspace graphics application owns, and memory accessible to the graphics card. The DRM functionality was later significantly enhanced to support additional ioctl operations including *modesetting* (KMS) - provided the card-specific “helper module” supported it.

KMS functionality is accessed via libdrm, as with all other functionality provided by the “drm core” kernel module. Functions and constants used to access KMS functionality via libdrm often have the `drm_` prefix.

The newer DRM drivers also:

- provide “buffer management” operations (usually via the GEM or TTM libraries) which are a higher-level of abstraction than the DMA-configuration APIs provided by DRM;
- implement a “modesetting” api which is a superset of the functionality available via the framebuffer modesetting ioctl;
- implement the “framebuffer” API;
- provide some card-specific logic that was previously implemented in the X DDX driver (such as the DDX driver *telling* the drm module what memory ranges may be mapped by users).

The user api for setting graphics modes is defined in libdrm file `xf86drmMode.h` (particularly, function `drmModeAttachMode` or `drmModeSetCrtc`). The kernel side of this api is done via the ioctls named `DRM_IOCTL_MODE_*`. There are some useful examples of modesetting linked to in the References section of this article.

The fact that some card-specific logic has moved from X to the kernel means that it is easier to implement alternatives to X (eg Wayland). The fact that buffer management code is in the kernel also makes it easier to implement alternatives to X.

The fact that a KMS driver is an integrated DRM *and* framebuffer driver allows smooth graphics handover from boot to X, and smooth/rapid switching between X and framebuffer-based virtual consoles.

Because some logic was moved from X to the kernel (esp. those where the userspace driver configures the drm kernel module), the corresponding IOCTLs are no longer useful. KMS drivers therefore provide implementations of these APIs which simply return an error-code.

Theoretically, KMS could also be implemented on non-Linux systems. However it does require porting or reimplementing the GEM/TTM modules, which is a non-trivial process. At the current time (late 2012) there are efforts underway to get KMS working on OpenBSD.

TTM and GEM

GEM is a kernel module that provides a library of functions for managing memory buffers for GPUs. Currently, it doesn’t handle on-card memory directly.

TTM performs the same purpose as GEM; it is more complex, but handles *on-card* memory better. TTM also implements the GEM API, so userspace doesn’t need to care which implementation is being used by the current driver.

DRM drivers expose a GEM interface via IOCTL operations, for the purpose of manipulating buffers associated with the graphics card that the driver handles.

DRM Module Implementation

The sourcecode for the drm core is in `drivers/gpu/drm`.

An example of a pure DRM driver can be found in `drivers/gpu/drm/tdfx` (supports 3dfx Voodoo cards). Another is in `drivers/gpu/drm/sis`.

Initialisation and Driver Registration

File `drm_drv.c` is the “entry point” for the drm module: it contains the `module_init` declaration, pointing to `drm_core_init()`.

```
drm_drv.c : on init, drm_core_init():
* initialises static var drm_minors_idr to hold an empty mapping of (minor->drm_device)
* creates & registers major-device-number DRM_MAJOR (226) with a very simple file_operations structure
* creates & registers a "class" object with sysfs (drm_class)
```

Card-specific DRM drivers have their own modules which define a list of PCI IDs for devices it can handle, then calls `drm_pci.c:drm_pci_init`.

For “old” drivers, `drm_pci_init` manually scans all known PCI devices, and calls `drm_get_pci_dev` for each matching device.

For “new” drivers, `drm_pci_init` instead expects the caller’s `pci_driver` table to include a pointer to a “probe” function; the PCI subsystem will call this for each matching device, and the probe function is expected to then call `drm_get_pci_dev`.

It appears that the change in the PCI subsystem from caller “scanning” to caller providing a “probe” callback happened *coincidentally* about the same time as the development of KMS, and therefore KMS drivers provide a probe method while older ones do not.

```
drm_pci.c:drm_get_pci_dev:
* allocates a device minor number in range 0..63 (LEGACY), and creates device node `/dev/dri/card{n}`
* optionally allocates a device minor number in range 64..127 (CONTROL), and creates device node `/dev/dri/controlD{n}` which is connected to the modesetting functions only
  (ie userspace code which can open this file can perform mode-setting but not generate graphics; can be helpful for unusual use-cases).
* allocates a drm device structure, and stores it in the drm_minors_idr map keyed by each of the allocated minor#s.
* calls the "load" function provided by the card-specific driver
```

Both the device nodes inherit the `file_operations` callbacks from their major device (created in `drm_core_init` earlier); this has a single hook for “open” operation, which points to `drm_stub_open`.

When a user opens the `/dev/dri/card{n}` file (for all operations) or the `/dev/dri/controlD{n}` file (for modesetting only), function `drm_stub.c:drm_stub_open` is passed the minor# of the opened file, and uses this as an index into the `drm_minors_idr` table to obtain a reference to the appropriate `drm_device` structure. It then stores this pointer into the open file structure so that future file operations can immediately locate the device-specific info.

The stub open function also does a magic switcheroo: it replaces the `file_operations` structure that the open file handle inherited from the `DRM_MAJOR` device with one provided by the card-specific driver. In effect, this bypasses the normal Unix behaviour where a major# references a driver and a minor-number indicates multiple instances of the same device, and instead allows each device minor# to point to a different driver. After the file-operations switch, all future file-related system calls performed by userspace go directly to the card-specific driver.

File Operations

As noted in the previous section, a card-specific driver can provide a table of `file_operations` callbacks to be invoked when the user does various operations on an open file-handle. For most drivers, the majority of entries in this table point back into common library functions provided by the drm core code.

IOCTLs

The standard list of IOCTL operations provided by a DRM driver can be seen in table `drm_drv.c:drm_ioctl`s.

Most drivers have a file-operations structure which maps the `ioctl` callback into the standard `drm_ioctl` library function. They then define an additional table of custom `ioctl` operations and store that in the `drm_driver` structure. The `drm_ioctl` function forwards all `ioctl` operations in range `COMMAND_BASE..COMMAND_END` (0x40-0xA0) using the device-specific table, and handles all others itself using its `ioctl`s table. This ensures that all the “standard” DRM `ioctl`s are implemented in the drm core, but card-specific drivers can implement additional `ioctl` operations that their corresponding X DDX driver presumably knows how to use.

Those entries in the main `drm_ioctl` table which are flagged with `DRM_ROOTONLY` return errors unless the caller is root (ie the X server itself). This allows the drm driver to separate calls into those safe for use by graphical client applications just doing direct drawing, and those that could be used to take over the machine (eg DMA configuration).

Those flagged with `DRM_MASTER` return errors unless the file handle used for the `ioctl` is marked as the “master” for the device. It is expected that the X server will be the first application to open the device node for any graphics device, and will immediately invoke the `SET_MASTER` `ioctl`. This allows it to “reserve” access to certain dangerous operations for itself. There is a corresponding `DROP_MASTER` `ioctl`. The operations flagged with `DRM_MASTER` presumably prevent security issues and race-conditions by restricting racy operations to the single “master” process.

As noted previously, the userspace API is defined in `libdrm`, not at the kernel level - and therefore there is no formal definition of what the various IOCTL operations do (at least none that I can find). The best way to find out the purpose of any specific IOCTL is to look at the `libdrm` source (which is somewhat better documented).

According to general DRI docs:

- there is a “per card lock to synchronize access”. Is this the `LOCK/UNLOCK` operations?
- the X server can specify what memory ranges client apps are permitted to map - is this the `ADD_MAP/RM_MAP` operations?

The DRI architecture in general expects clients to ask X to allocate data buffers for sharing with the card, and for X to return the addresses of the buffers. Maybe the `ADD_BUFS/MARK_BUFS` operations are for this? Particularly as `ADD_BUFS` is marked with `DRM_ROOTONLY` (ie X), but `FREE_BUFS` is not (ie the client can free the buffer itself).

I also seem to remember hearing about the necessity of validating the command-stream instructions to prevent system takeover or even damage. Are these the ?? operations?

Perhaps the CTX operations are used to support multiple applications accessing the ring-buffer at the same time?

DRM Driver Source Code

The source for these drivers lives in subdirs of `drivers/gpu/drm`. These register themselves with the drm core module, and the file-operations switcheroo occurs when userspace opens the appropriate file.

For AMD cards, the KMS driver source is in `drivers/gpu/drm/radeon` and the compiled module is in `/lib/modules/{version}/kernel/drivers/gpu/drm/radeon/radeon.ko`. This driver supports all the different models of AMD/ATI cards, even though they are quite different.

The X server has a different KMS-enabled userspace driver for each graphics card that has a kernel DRM driver; like the X Native DDX drivers, these drivers are responsible for implementing all acceleration themselves (ie performance depends heavily on the quality of this driver). However unlike the X Native DDX drivers, they can leave modesetting and buffer management up to the kernel layer (which can do it more efficiently). In addition, the fact that buffers are allocated in the kernel opens more options for applications to access the graphics card directly (bypassing X for performance). The X native DDX driver and the X KMS-enabled DDX driver for a card often share significant amounts of code related to generating “commands” for drawing operations.

Note that although a DRM driver implements the framebuffer API, there may also be a separate (simpler) framebuffer driver available for the same card. This is true for some intel and nvidia cards for example - while AMD have just a single combined driver. When the KMS driver is active, the alternative framebuffer driver will not be used. However having the option of using the simpler framebuffer is a good thing (eg embedded systems without requirement for high-performance graphics, or servers); of course this does mean maintaining duplicated code though, so some graphics cards do not have a separate framebuffer driver anymore.

Initialisation and Driver Registration

```
radeon_drv.c on init:
* registers itself with the PCI bus as handler for specific PCI IDs, using
  + driver=kms_driver
  + pci_driver = radeon_kms_pci_driver

The PCI bus then calls back into radeon_pci_probe when a radeon card is found.
This calls drm_get_pci_dev which:
* calls drm_get_minor(...,DRM_MINOR_CONTROL) which registers `/dev/dri/controlD{n}`
* calls drm_get_minor(...,DRM_MINOR_LEGACY) which registers `/dev/dri/card{n}`
* calls back into dev->driver->load, ie radeon_driver_load_kms.

drm/drm_stub.c: drm_get_minor
+ calls drm_minor_get_id to allocate an unused minor-device-number:
  + type=DRM_MINOR_LEGACY: range = 0..63
  + type=DRM_MINOR_CONTROL: range = 64..127
  + type=DRM_MINOR_RENDER: range=128..255 (TODO: is this used anymore?)
  + this also ensures that drm_minors_idr has enough space for the new node [NOTE: UGLY SIDE-EFFECT]
+ allocates space for a "drm_minor", ie a simple wrapper around card-specific drm_device
+ sets the drm_minor structure to point to the drm_device for the radeon card
+ stores the new drm_minor structure into the drm_minors_idr
+ calls drm_sysfs_device_add

drm_sysfs_device_add:
+ initialises some kdev-related fields
+ sets device_name to either `/dev/dri/controlD{n}`, `/dev/dri/renderD{n}` or `/dev/dri/card{n}` depending on whether
  the type param was DRM_MINOR_CONTROL/DRM_MINOR_RENDER/DRM_MINOR_LEGACY
+ calls device_register

radeon_driver_load_kms:
* calls radeon_device_init (inits non-display parts of card, eg command-processor)
* calls radeon_modeset_init (inits display parts, eg CRTCs, encoders)
```

```
* calls radeon_acpi_init (configures ACPI)
// ...
}
```

The different card models are handled by initialising the various callback tables appropriately, eg if the card is a “southern islands” card, then the tables are defined in file `si.c`.

Opening of the Device File

Userspace eventually opens `/dev/dri/card{N}`. Function `drm_stub_open()` is executed, which:

- uses `drm_minors_idr` to map the minor device number to a struct `drm_device` registered by the radeon driver on module initialisation
- then does a switchover on the `file_operations` structure from that `drm_device` so the radeon-provided one is used directly for future file operations performed by userspace.

IOCTL operations

Userspace then does IOCTLs on the filehandle, which are forwarded to `radeon_ioctl_kms`.

File `radeon_kms.c` defines structure `radeon_ioctl_kms` to define all the IOCTL operations available to userspace. These IOCTLs are not “standardised”; they are understood only by the matching libdrm file (eg `/usr/lib/i386-linux-gnu/libdrm_radeon.so.1`).

The first block (up until the “KMS” comment) are obsolete/disabled IOCTLs that just return `-EINVAL`. ?? who uses these ??

The following IOCTLs are the APIs used by kms-enabled graphics drivers:

- `INFO`
- `CS` – submit a batch of command-stream instructions
- `GEM_INFO` – ??
- `GEM_CREATE` – create a buffer
- `GEM_MMAP` – map a GEM buffer into the caller’s address-space
- `GEM_SET_DOMAIN` –
- `GEM_READ` –
- `GEM_PWRITE` –
- `GEM_WAIT_IDLE` –
- `GEM_SET_TILING` –
- `GEM_GET_TILING` –
- `GEM_BUSY` –
- `GEM_VA` –

Ring Buffer Handling

One of the important tasks that userspace apps need to do is send “command stream” instructions direct to the graphics card (not via X). The radeon graphics cards use a “ring buffer” structure to provide a high-performance data channel for these instructions.

File `radeon_ring.c` implements the necessary logic. In particular:

```
+ `radeon_ib_get` --> returns one of the "indirect" buffers - in r5xx there are two.
+ `radeon_ib_schedule` --> emits a command on the circular buffer that tells the GPU to use an IB buffer.
```

Command Stream Parsing

One of the important tasks that userspace apps need to do is send “command stream” instructions direct to the graphics card (not via X). However some of these commands can be used to take over the host computer or even damage the graphics card or display. Therefore the commands sent are first “validated” by the KMS driver before being passed on.

File `radeon_cs_parse.c` implements the necessary logic. In particular:

```
radeon_cs_ib_chunk():
+ obtains an intermediate buffer
+ calls `radeon_sc_parse`
+ calls `radeon_cs_finish_pages`
+ calls `radeon_cs_sync_rings`
+ calls `radeon_ib_schedule`
==> writes commands to the main ring to load the intermediate buffer now

radeon_cs.c:radeon_cs_ioctl
+ validates commands then passes them on to ring buffer
+ referenced from radeon_kms.c as
  DRM_IOCTL_DEF_DRV(
```

Questions

- Is `DRM_MINOR_RENDER` totally obsolete? – looks like it.
- Does the “`/dev/drm`” file referenced in comments exist anymore – doesn’t look like it. ?? does `/dev/drm` appear if kernel is booted with `nomodeset`?

References

Information in this article has been pulled from *many* sources. Here are a few of them:

- [DRI for Beginners](#)
- [Example of initialising graphics mode using the libdrm api](#)
- [David Hermann’s libdrm modesetting example](#)

And here are some other useful references:


- [Pekka Paalanen’s History of Graphics Modesetting in X11, KMS and DRM](#)
- [A project to document DRM/KMS](#) and [more here](#)

Footnotes

1. Consoles (such as the boot console) were output using BIOS VGA text operations, and did not support graphics. ↩

0 Comments Mine of Information Login

Recommend Tweet Share Sort by Oldest



LOG IN WITH

OR SIGN UP WITH DISQUS ?

Be the first to comment.

1 comment • 2 years ago

Vasu Jinagam — Good one :)

Mine of Information - Symmetric Encryption in Java

2 comments • 6 years ago

Pepe LeVamp — amazing stuff. very honest and to the point. it -is- confusing & complex using the java APIs for this stuff.

1 comment • 3 years ago

bluejumper — I found this to be a very informative read; thank you.

Mine of Information - Downloading Jenkins

2 comments • 2 years ago

Simon Kitching — Ah - how embarrassing. I didn't understand how that menu/button combo worked - when hovering over the "menu", a drop-down list with no war-option is shown. But clicking on the button aka menu-header itself does download the warfile. Normally, a drop-down menu includes all possible ...

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy