Userland interfaces

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

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

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

libdrm Device Lookup

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 22)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_ioctl.c
   :doc: getunique and setversion story
```

Primary Nodes, DRM Master and Authentication

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 31)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_auth.c
   :doc: master and authentication
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 34)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_auth.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 37)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/drm/drm_auth.h
   :internal:
```

DRM Display Resource Leasing

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 46)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_lease.c
   :doc: drm leasing
```

Open-Source Userspace Requirements

The DRM subsystem has stricter requirements than most other kernel subsystems on what the userspace side for new uAPI needs to look like. This section here explains what exactly those requirements are, and why they exist.

The short summary is that any addition of DRM uAPI requires corresponding open-sourced userspace patches, and those patches must be reviewed and ready for merging into a suitable and canonical upstream project.

GFX devices (both display and render/GPU side) are really complex bits of hardware, with userspace and kernel by necessity having to work together really closely. The interfaces, for rendering and modesetting, must be extremely wide and flexible, and therefore it is almost always impossible to precisely define them for every possible corner case. This in turn makes it really practically infeasible to differentiate between behaviour that's required by userspace, and which must not be changed to avoid regressions, and behaviour which is only an accidental artifact of the current implementation.

Without access to the full source code of all userspace users that means it becomes impossible to change the implementation details, since userspace could depend upon the accidental behaviour of the current implementation in minute details. And debugging such regressions without access to source code is pretty much impossible. As a consequence this means:

- The Linux kernel's "no regression" policy holds in practice only for open-source userspace of the DRM subsystem. DRM developers are perfectly fine if closed-source blob drivers in userspace use the same uAPI as the open drivers, but they must do so in the exact same way as the open drivers. Creative (ab)use of the interfaces will, and in the past routinely has, lead to breakage.
- Any new userspace interface must have an open-source implementation as demonstration vehicle.

The other reason for requiring open-source userspace is uAPI review. Since the kernel and userspace parts of a GFX stack must work together so closely, code review can only assess whether a new interface achieves its goals by looking at both sides. Making sure that the interface indeed covers the use-case fully leads to a few additional requirements:

- The open-source userspace must not be a toy/test application, but the real thing. Specifically it needs to handle all the usual error and corner cases. These are often the places where new uAPI falls apart and hence essential to assess the fitness of a proposed interface.
- The userspace side must be fully reviewed and tested to the standards of that userspace project. For e.g. mesa this means piglit testcases and review on the mailing list. This is again to ensure that the new interface actually gets the job done. The userspace-side reviewer should also provide an Acked-by on the kernel uAPI patch indicating that they believe the proposed uAPI is sound and sufficiently documented and validated for userspace's consumption.
- The userspace patches must be against the canonical upstream, not some vendor fork. This is to make sure that no one cheats on the review and testing requirements by doing a quick fork.
- The kernel patch can only be merged after all the above requirements are met, but it **must** be merged to either drm-next or drm-misc-next **before** the userspace patches land. uAPI always flows from the kernel, doing things the other way round risks divergence of the uAPI definitions and header files.

These are fairly steep requirements, but have grown out from years of shared pain and experience with uAPI added hastily, and almost always regretted about just as fast. GFX devices change really fast, requiring a paradigm shift and entire new set of uAPI interfaces every few years at least. Together with the Linux kernel's guarantee to keep existing userspace running for 10+ years this is already rather painful for the DRM subsystem, with multiple different uAPIs for the same thing co-existing. If we add a few more complete mistakes into the mix every year it would be entirely unmanageable.

Render nodes

DRM core provides multiple character-devices for user-space to use. Depending on which device is opened, user-space can perform a different set of operations (mainly ioctls). The primary node is always created and called card<num>. Additionally, a currently unused control node, called controlD<num> is also created. The primary node provides all legacy operations and historically was the only interface used by userspace. With KMS, the control node was introduced. However, the planned KMS control interface has never been written and so the control node stays unused to date.

With the increased use of offscreen renderers and GPGPU applications, clients no longer require running compositors or graphics servers to make use of a GPU. But the DRM API required unprivileged clients to authenticate to a DRM-Master prior to getting GPU access. To avoid this step and to grant clients GPU access without authenticating, render nodes were introduced. Render nodes solely serve render clients, that is, no modesetting or privileged ioctls can be issued on render nodes. Only non-global rendering commands are allowed. If a driver supports render nodes, it must advertise it via the DRIVER_RENDER DRM driver capability. If not supported, the primary node must be used for render clients together with the legacy drmAuth authentication procedure.

If a driver advertises render node support, DRM core will create a separate render node called renderD<num>. There will be one render node per device. No ioctls except PRIME-related ioctls will be allowed on this node. Especially GEM_OPEN will be explicitly prohibited. Render nodes are designed to avoid the buffer-leaks, which occur if clients guess the flink names or mmap offsets on the legacy interface. Additionally to this basic interface, drivers must mark their driver-dependent render-only ioctls as DRM_RENDER_ALLOW so render clients can use them. Driver authors must be careful not to allow any privileged ioctls on render nodes.

With render nodes, user-space can now control access to the render node via basic file-system access-modes. A running graphics server which authenticates clients on the privileged primary/legacy node is no longer required. Instead, a client can open the render node and is immediately granted GPU access. Communication between clients (or servers) is done via PRIME. FLINK from render node to legacy node is not supported. New clients must not use the insecure FLINK interface.

Besides dropping all modeset/global ioctls, render nodes also drop the DRM-Master concept. There is no reason to associate render clients with a DRM-Master as they are independent of any graphics server. Besides, they must work without any running master,

anyway. Drivers must be able to run without a master object if they support render nodes. If, on the other hand, a driver requires shared state between clients which is visible to user-space and accessible beyond open-file boundaries, they cannot support render nodes.

Device Hot-Unplug

> **Note**
>
> The following is the plan. Implementation is not there yet (2020 May).

Graphics devices (display and/or render) may be connected via USB (e.g. display adapters or docking stations) or Thunderbolt (e.g. eGPU). An end user is able to hot-unplug this kind of devices while they are being used, and expects that the very least the machine does not crash. Any damage from hot-unplugging a DRM device needs to be limited as much as possible and userspace must be given the chance to handle it if it wants to. Ideally, unplugging a DRM device still lets a desktop continue to run, but that is going to need explicit support throughout the whole graphics stack: from kernel and userspace drivers, through display servers, via window system protocols, and in applications and libraries.

Other scenarios that should lead to the same are: unrecoverable GPU crash, PCI device disappearing off the bus, or forced unbind of a driver from the physical device.

In other words, from userspace perspective everything needs to keep on working more or less, until userspace stops using the disappeared DRM device and closes it completely. Userspace will learn of the device disappearance from the device removed uevent, ioctls returning ENODEV (or driver-specific ioctls returning driver-specific things), or open() returning ENXIO.

Only after userspace has closed all relevant DRM device and dmabuf file descriptors and removed all mmaps, the DRM driver can tear down its instance for the device that no longer exists. If the same physical device somehow comes back in the mean time, it shall be a new DRM device.

Similar to PIDs, chardev minor numbers are not recycled immediately. A new DRM device always picks the next free minor number compared to the previous one allocated, and wraps around when minor numbers are exhausted.

The goal raises at least the following requirements for the kernel and drivers.

Requirements for KMS UAPI

- KMS connectors must change their status to disconnected.
- Legacy modesets and pageflips, and atomic commits, both real and TEST_ONLY, and any other ioctls either fail with ENODEV or fake success.
- Pending non-blocking KMS operations deliver the DRM events userspace is expecting. This applies also to ioctls that faked success.
- open() on a device node whose underlying device has disappeared will fail with ENXIO.
- Attempting to create a DRM lease on a disappeared DRM device will fail with ENODEV. Existing DRM leases remain and work as listed above.

Requirements for Render and Cross-Device UAPI

- All GPU jobs that can no longer run must have their fences force-signalled to avoid inflicting hangs on userspace. The associated error code is ENODEV.
- Some userspace APIs already define what should happen when the device disappears (OpenGL, GL ES: GL_KHR_robustness; Vulkan: VK_ERROR_DEVICE_LOST; etc.). DRM drivers are free to implement this behaviour the way they see best, e.g. returning failures in driver-specific ioctls and handling those in userspace drivers, or rely on uevents, and so on.
- dmabuf which point to memory that has disappeared will either fail to import with ENODEV or continue to be successfully imported if it would have succeeded before the disappearance. See also about memory maps below for already imported dmabufs.
- Attempting to import a dmabuf to a disappeared device will either fail with ENODEV or succeed if it would have succeeded without the disappearance.
- open() on a device node whose underlying device has disappeared will fail with ENXIO.

Requirements for Memory Maps

Memory maps have further requirements that apply to both existing maps and maps created after the device has disappeared. If the underlying memory disappears, the map is created or modified such that reads and writes will still complete successfully but the result is undefined. This applies to both userspace mmap()'d memory and memory pointed to by dmabuf which might be mapped to other devices (cross-device dmabuf imports).

Raising SIGBUS is not an option, because userspace cannot realistically handle it. Signal handlers are global, which makes them extremely difficult to use correctly from libraries like those that Mesa produces. Signal handlers are not composable, you can't have different handlers for GPU1 and GPU2 from different vendors, and a third handler for mmapped regular files. Threads cause additional pain with signal handling as well.

IOCTL Support on Device Nodes

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 291)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_ioctl.c
   :doc: driver specific ioctls
```

Recommended IOCTL Return Values

In theory a driver's IOCTL callback is only allowed to return very few error codes. In practice it's good to abuse a few more. This section documents common practice within the DRM subsystem:

ENOENT:

Strictly this should only be used when a file doesn't exist e.g. when calling the open() syscall. We reuse that to signal any kind of object lookup failure, e.g. for unknown GEM buffer object handles, unknown KMS object handles and similar cases.

ENOSPC:

Some drivers use this to differentiate "out of kernel memory" from "out of VRAM". Sometimes also applies to other limited gpu resources used for rendering (e.g. when you have a special limited compression buffer). Sometimes resource allocation/reservation issues in command submission IOCTLs are also signalled through EDEADLK.

Simply running out of kernel/system memory is signalled through ENOMEM.

EPERM/EACCES:

Returned for an operation that is valid, but needs more privileges. E.g. root-only or much more common, DRM master-only operations return this when called by unpriviledged clients. There's no clear difference between EACCES and EPERM.

ENODEV:

The device is not present anymore or is not yet fully initialized.

EOPNOTSUPP:

Feature (like PRIME, modesetting, GEM) is not supported by the driver.

ENXIO:

Remote failure, either a hardware transaction (like i2c), but also used when the exporting driver of a shared dma-buf or fence doesn't support a feature needed.

EINTR:

DRM drivers assume that userspace restarts all IOCTLs. Any DRM IOCTL can return EINTR and in such a case should be restarted with the IOCTL parameters left unchanged.

EIO:

The GPU died and couldn't be resurrected through a reset. Modesetting hardware failures are signalled through the "link status" connector property.

EINVAL:

Catch-all for anything that is an invalid argument combination which cannot work.

IOCTL also use other error codes like ETIME, EFAULT, EBUSY, ENOTTY but their usage is in line with the common meanings. The above list tries to just document DRM specific patterns. Note that ENOTTY has the slightly unintuitive meaning of "this IOCTL does not exist", and is used exactly as such in DRM.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 352)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/drm/drm_ioctl.h
   :internal:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 355)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_ioctl.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 358)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_ioc32.c
   :export:
```

Testing and validation

Testing Requirements for userspace API

New cross-driver userspace interface extensions, like new IOCTL, new KMS properties, new files in sysfs or anything else that constitutes an API change should have driver-agnostic testcases in IGT for that feature, if such a test can be reasonably made using IGT for the target hardware.

Validating changes with IGT

There's a collection of tests that aims to cover the whole functionality of DRM drivers and that can be used to check that changes to DRM drivers or the core don't regress existing functionality. This test suite is called IGT and its code and instructions to build and run can be found in https://gitlab.freedesktop.org/drm/igt-gpu-tools/.

Using VKMS to test DRM API

VKMS is a software-only model of a KMS driver that is useful for testing and for running compositors. VKMS aims to enable a virtual display without the need for a hardware display capability. These characteristics made VKMS a perfect tool for validating the DRM core behavior and also support the compositor developer. VKMS makes it possible to test DRM functions in a virtual machine without display, simplifying the validation of some of the core changes.

To Validate changes in DRM API with VKMS, start setting the kernel: make sure to enable VKMS module; compile the kernel with the VKMS enabled and install it in the target machine. VKMS can be run in a Virtual Machine (QEMU, virtme or similar). It's recommended the use of KVM with the minimum of 1GB of RAM and four cores.

It's possible to run the IGT-tests in a VM in two ways:

1. Use IGT inside a VM
2. Use IGT from the host machine and write the results in a shared directory.

As follow, there is an example of using a VM with a shared directory with the host machine to run igt-tests. As an example it's used virtme:

```
$ virtme-run --rwdir /path/for/shared_dir --kdir=path/for/kernel/directory --mods=auto
```

Run the igt-tests in the guest machine, as example it's ran the 'kms_flip' tests:

```
$ /path/for/igt-gpu-tools/scripts/run-tests.sh -p -s -t "kms_flip.*" -v
```

In this example, instead of build the igt_runner, Piglit is used (-p option); it's created html summary of the tests results and it's saved in the folder "igt-gpu-tools/results"; it's executed only the igt-tests matching the -t option.

Display CRC Support

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 421)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_debugfs_crc.c
   :doc: CRC ABI
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 424)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_debugfs_crc.c
   :export:
```

Debugfs Support

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 430)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/drm/drm_debugfs.h
   :internal:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 433)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_debugfs.c
   :export:
```

Sysfs Support

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 439)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_sysfs.c
   :doc: overview
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 442)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/drm_sysfs.c
   :export:
```

VBlank event handling

The DRM core exposes two vertical blank related ioctls:

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

DRM_IOCTL_MODESET_CTL
: This was only used for user-mode-settind drivers around modesetting changes to allow the kernel to update the vblank interrupt after mode setting, since on many devices the vertical blank counter is reset to 0 at some point during modeset. Modern drivers should not call this any more since with kernel mode setting it is a no-op.

Userspace API Structures

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 466)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/uapi/drm/drm_mode.h
   :doc: overview
```

CRTC index

CRTC's have both an object ID and an index, and they are not the same thing. The index is used in cases where a densely packed identifier for a CRTC is needed, for instance a bitmask of CRTC's. The member possible_crtcs of struct drm_mode_get_plane is an example.

DRM_IOCTL_MODE_GETRESOURCES populates a structure with an array of CRTC ID's, and the CRTC index is its position in this array.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 482)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/uapi/drm/drm.h
   :internal:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]drm-uapi.rst, line 485)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/uapi/drm/drm_mode.h
   :internal:
```