
Linux Gpu Documentation

The kernel development community

Jun 10, 2024

CONTENTS

1	Introduction	1
1.1	Style Guidelines	1
1.2	Getting Started	2
1.3	Contribution Process	2
1.4	Simple DRM drivers to use as examples	3
1.5	External References	3
2	DRM Internals	5
2.1	Driver Initialization	5
2.2	Open/Close, File Operations and IOCTLs	31
2.3	Misc Utilities	43
2.4	Unit testing	51
2.5	Legacy Support Code	51
3	DRM Memory Management	53
3.1	The Translation Table Manager (TTM)	53
3.2	The Graphics Execution Manager (GEM)	71
3.3	VMA Offset Manager	116
3.4	PRIME Buffer Sharing	123
3.5	DRM MM Range Allocator	131
3.6	DRM GPU VA Manager	143
3.7	DRM Buddy Allocator	171
3.8	DRM Cache Handling and Fast WC memcpy()	174
3.9	DRM Sync Objects	175
3.10	DRM Execution context	181
3.11	GPU Scheduler	187
4	Kernel Mode Setting (KMS)	205
4.1	Overview	205
4.2	KMS Core Structures and Functions	208
4.3	Modeset Base Object Abstraction	222
4.4	Atomic Mode Setting	226
4.5	CRTC Abstraction	258
4.6	Frame Buffer Abstraction	282
4.7	DRM Format Handling	289
4.8	Dumb Buffer Objects	297
4.9	Plane Abstraction	297
4.10	Display Modes Function Reference	321
4.11	Connector Abstraction	341

4.12	Encoder Abstraction	392
4.13	KMS Locking	399
4.14	KMS Properties	405
4.15	Vertical Blanking	442
4.16	Vertical Blank Work	455
5	Mode Setting Helper Functions	459
5.1	Modeset Helper Reference for Common Vtables	459
5.2	Atomic Modeset Helper Functions Reference	480
5.3	Simple KMS Helper Reference	518
5.4	fbdev Helper Functions Reference	523
5.5	format Helper Functions Reference	535
5.6	Framebuffer DMA Helper Functions Reference	544
5.7	Framebuffer GEM Helper Reference	545
5.8	Bridges	551
5.9	Panel Helper Reference	577
5.10	Panel Self Refresh Helper Reference	585
5.11	HDCP Helper Functions Reference	586
5.12	Display Port Helper Functions Reference	588
5.13	Display Port CEC Helper Functions Reference	618
5.14	Display Port Dual Mode Adaptor Helper Functions Reference	619
5.15	Display Port MST Helpers	623
5.16	MIPI DBI Helper Functions Reference	652
5.17	MIPI DSI Helper Functions Reference	663
5.18	Display Stream Compression Helper Functions Reference	678
5.19	Output Probing Helper Functions Reference	689
5.20	EDID Helper Functions Reference	695
5.21	SCDC Helper Functions Reference	710
5.22	HDMI Infoframes Helper Reference	713
5.23	Rectangle Utilities Reference	723
5.24	Flip-work Helper Reference	730
5.25	Auxiliary Modeset Helpers	732
5.26	OF/DT Helpers	734
5.27	Legacy Plane Helper Reference	739
5.28	Legacy CRTC/Modeset Helper Functions Reference	741
5.29	Privacy-screen class	745
6	Userland interfaces	753
6.1	libdrm Device Lookup	753
6.2	Primary Nodes, DRM Master and Authentication	754
6.3	DRM Display Resource Leasing	757
6.4	Open-Source Userspace Requirements	757
6.5	Render nodes	759
6.6	Device Hot-Unplug	760
6.7	IOCTL Support on Device Nodes	761
6.8	Testing and validation	767
6.9	Sysfs Support	773
6.10	VBlank event handling	774
6.11	Userspace API Structures	775
7	DRM client usage stats	797
7.1	File format specification	797

7.2	Implementation Details	800
8	DRM Driver uAPI	801
8.1	drm/i915 uAPI	801
8.2	drm/nouveau uAPI	833
9	Kernel clients	839
10	GPU Driver Documentation	847
10.1	drm/amdgpu AMDgpu driver	847
10.2	drm/i915 Intel GFX Driver	963
10.3	drm/mcde ST-Ericsson MCDE Multi-channel display engine	1092
10.4	drm/meson AmLogic Meson Video Processing Unit	1093
10.5	drm/pl111 ARM PrimeCell PL110 and PL111 CLCD Driver	1097
10.6	drm/tegra NVIDIA Tegra GPU and display driver	1097
10.7	drm/tve200 Faraday TV Encoder 200	1108
10.8	drm/v3d Broadcom V3D Graphics Driver	1108
10.9	drm/vc4 Broadcom VC4 Graphics Driver	1109
10.10	drm/vkms Virtual Kernel Modesetting	1113
10.11	drm/bridge/dw-hdmi Synopsys DesignWare HDMI Controller	1116
10.12	drm/xen-front Xen para-virtualized frontend driver	1118
10.13	Arm Framebuffer Compression (AFBC)	1119
10.14	drm/komeda Arm display driver	1122
11	Backlight support	1143
12	VGA Switcheroo	1153
12.1	Modes of Use	1154
12.2	API	1155
12.3	Handlers	1164
13	VGA Arbiter	1167
13.1	vgaarb kernel/userspace ABI	1167
13.2	In-kernel interface	1168
13.3	libpciaccess	1171
13.4	xf86VGAArbiter (X server implementation)	1173
13.5	References	1173
14	Automated testing of the DRM subsystem	1175
14.1	Introduction	1175
14.2	Relevant files	1175
14.3	How to enable automated testing on your tree	1176
14.4	How to update test expectations	1176
14.5	How to expand coverage	1177
14.6	How to test your changes to the scripts	1177
14.7	How to incorporate external fixes in your testing	1177
14.8	How to deal with automated testing labs that may be down	1177
15	TODO list	1179
15.1	Difficulty	1179
15.2	Remove custom dumb_map_offset implementations	1179
15.3	Convert existing KMS drivers to atomic modesetting	1180
15.4	Clean up the clipped coordination confusion around planes	1180

15.5	Improve plane atomic_check helpers	1180
15.6	Convert early atomic drivers to async commit helpers	1181
15.7	Fallout from atomic KMS	1181
15.8	Get rid of dev->struct_mutex from GEM drivers	1181
15.9	Move Buffer Object Locking to dma_resv_lock()	1182
15.10	Convert logging to drm_* functions with drm_device parameter	1182
15.11	Convert drivers to use simple modeset suspend/resume	1182
15.12	Convert drivers to use drm_fbdev_generic_setup()	1183
15.13	Reimplement functions in drm_fbdev_fb_ops without fbdev	1183
15.14	Benchmark and optimize blitting and format-conversion function	1183
15.15	drm_framebuffer_funcs and drm_mode_config_funcs.fb_create cleanup	1183
15.16	Generic fbdev defio support	1184
15.17	connector register/unregister fixes	1184
15.18	Remove load/unload callbacks	1185
15.19	Replace drm_detect_hdmi_monitor() with drm_display_info.is_hdmi	1185
15.20	Consolidate custom driver modeset properties	1185
15.21	Use struct iosys_map throughout codebase	1186
15.22	Review all drivers for setting struct drm_mode_config.{max_width,max_height} correctly	1186
15.23	Request memory regions in all drivers	1186
15.24	Remove driver dependencies on FB_DEVICE	1187
15.25	Clean up checks for already prepared/enabled in panels	1187
15.26	Make panic handling work	1187
15.27	Clean up the debugfs support	1188
15.28	Object lifetime fixes	1189
15.29	Remove automatic page mapping from dma-buf importing	1189
15.30	Add unit tests using the Kernel Unit Testing (KUnit) framework	1189
15.31	Enable trinity for DRM	1190
15.32	Make KMS tests in i-g-t generic	1190
15.33	Extend virtual test driver (VKMS)	1190
15.34	Backlight Refactoring	1190
15.35	AMD DC Display Driver	1190
15.36	Convert fbdev drivers to DRM	1192
16	GPU RFC Section	1193
16.1	I915 DG1/LMEM RFC Section	1193
16.2	I915 GuC Submission/DRM Scheduler Section	1194
16.3	I915 Small BAR RFC Section	1199
16.4	I915 VM_BIND feature design and use cases	1203
16.5	Xe - Merge Acceptance Plan	1213
	Index	1219

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]

1.1 Style Guidelines

For consistency this documentation uses American English. Abbreviations are written as all-uppercase, for example: DRM, KMS, IOCTL, CRTC, and so on. To aid in reading, documentations make full use of the markup characters kerneldoc provides: @parameter for function parameters, @member for structure members (within the same structure), &struct structure to reference structures and function() for functions. These all get automatically hyperlinked if kerneldoc for the referenced objects exists. When referencing entries in function vtables (and structure members in general) please use &vtable_name.vfunc. Unfortunately this does not yet yield a direct link to the member, only the structure.

Except in special situations (to separate locked from unlocked variants) locking requirements for functions aren't documented in the kerneldoc. Instead locking should be checked at runtime using e.g. `WARN_ON(!mutex_is_locked(...));`. Since it's much easier to ignore documentation than runtime noise this provides more value. And on top of that runtime checks do need to be updated when the locking rules change, increasing the chances that they're correct. Within the documentation the locking rules should be explained in the relevant structures: Either in the comment for the lock explaining what it protects, or data fields need a note about which lock protects them, or both.

Functions which have a non-void return value should have a section called "Returns" explaining the expected return values in different cases and their meanings. Currently there's no consensus whether that section name should be all upper-case or not, and whether it should end in a colon or not. Go with the file-local style. Other common section names are "Notes" with information for dangerous or tricky corner cases, and "FIXME" where the interface could be cleaned up.

Also read the guidelines for the kernel documentation at large.

1.1.1 Documentation Requirements for kAPI

All kernel APIs exported to other modules must be documented, including their datastructures and at least a short introductory section explaining the overall concepts. Documentation should be put into the code itself as kerneldoc comments as much as reasonable.

Do not blindly document everything, but document only what's relevant for driver authors: Internal functions of `drm.ko` and definitely static functions should not have formal kerneldoc comments. Use normal C comments if you feel like a comment is warranted. You may use kerneldoc syntax in the comment, but it shall not start with a `/**` kerneldoc marker. Similar for data structures, annotate anything entirely private with `/* private: */` comments as per the documentation guide.

1.2 Getting Started

Developers interested in helping out with the DRM subsystem are very welcome. Often people will resort to sending in patches for various issues reported by checkpatch or sparse. We welcome such contributions.

Anyone looking to kick it up a notch can find a list of janitorial tasks on the [TODO list](#).

1.3 Contribution Process

Mostly the DRM subsystem works like any other kernel subsystem, see the main process guidelines and documentation for how things work. Here we just document some of the specialities of the GPU subsystem.

1.3.1 Feature Merge Deadlines

All feature work must be in the `linux-next` tree by the `-rc6` release of the current release cycle, otherwise they must be postponed and can't reach the next merge window. All patches must have landed in the `drm-next` tree by latest `-rc7`, but if your branch is not in `linux-next` then this must have happened by `-rc6` already.

After that point only bugfixes (like after the upstream merge window has closed with the `-rc1` release) are allowed. No new platform enabling or new drivers are allowed.

This means that there's a blackout-period of about one month where feature work can't be merged. The recommended way to deal with that is having a `-next` tree that's always open, but making sure to not feed it into `linux-next` during the blackout period. As an example, `drm-misc` works like that.

1.3.2 Code of Conduct

As a freedesktop.org project, dri-devel, and the DRM community, follows the Contributor Covenant, found at: <https://www.freedesktop.org/wiki/CodeOfConduct>

Please conduct yourself in a respectful and civilised manner when interacting with community members on mailing lists, IRC, or bug trackers. The community represents the project as a whole, and abusive or bullying behaviour is not tolerated by the project.

1.4 Simple DRM drivers to use as examples

The DRM subsystem contains a lot of helper functions to ease writing drivers for simple graphic devices. For example, the *drivers/gpu/drm/tiny/* directory has a set of drivers that are simple enough to be implemented in a single source file.

These drivers make use of the `struct drm_simple_display_pipe_funcs`, that hides any complexity of the DRM subsystem and just requires drivers to implement a few functions needed to operate the device. This could be used for devices that just need a display pipeline with one full-screen scanout buffer feeding one output.

The tiny DRM drivers are good examples to understand how DRM drivers should look like. Since are just a few hundreds lines of code, they are quite easy to read.

1.5 External References

Delving into a Linux kernel subsystem for the first time can be an overwhelming experience, one needs to get familiar with all the concepts and learn about the subsystem's internals, among other details.

To shallow the learning curve, this section contains a list of presentations and documents that can be used to learn about DRM/KMS and graphics in general.

There are different reasons why someone might want to get into DRM: porting an existing fbdev driver, write a DRM driver for a new hardware, fixing bugs that could face when working on the graphics user-space stack, etc. For this reason, the learning material covers many aspects of the Linux graphics stack. From an overview of the kernel and user-space stacks to very specific topics.

The list is sorted in reverse chronological order, to keep the most up-to-date material at the top. But all of them contain useful information, and it can be valuable to go through older material to understand the rationale and context in which the changes to the DRM subsystem were made.

1.5.1 Conference talks

- [An Overview of the Linux and Userspace Graphics Stack](#) - Paul Kocialkowski (2020)
- [Getting pixels on screen on Linux: introduction to Kernel Mode Setting](#) - Simon Ser (2020)
- [Everything Great about Upstream Graphics](#) - Daniel Vetter (2019)
- [An introduction to the Linux DRM subsystem](#) - Maxime Ripard (2017)
- [Embrace the Atomic \(Display\) Age](#) - Daniel Vetter (2016)
- [Anatomy of an Atomic KMS Driver](#) - Laurent Pinchart (2015)
- [Atomic Modesetting for Drivers](#) - Daniel Vetter (2015)
- [Anatomy of an Embedded KMS Driver](#) - Laurent Pinchart (2013)

1.5.2 Slides and articles

- [Understanding the Linux Graphics Stack](#) - Bootlin (2022)
- [DRM KMS overview](#) - STMicroelectronics (2021)
- [Linux graphic stack](#) - Nathan Gauër (2017)
- [Atomic mode setting design overview, part 1](#) - Daniel Vetter (2015)
- [Atomic mode setting design overview, part 2](#) - Daniel Vetter (2015)
- [The DRM/KMS subsystem from a newbie's point of view](#) - Boris Brezillon (2014)
- [A brief introduction to the Linux graphics stack](#) - Iago Toral (2014)
- [The Linux Graphics Stack](#) - Jasper St. Pierre (2012)

DRM INTERNALS

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

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

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

2.1 Driver Initialization

At the core of every DRM driver is a `struct drm_driver` structure. Drivers typically statically initialize a `drm_driver` structure, and then pass it to `drm_dev_alloc()` to allocate a device instance. After the device instance is fully initialized it can be registered (which makes it accessible from userspace) using `drm_dev_register()`.

The `struct 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 `struct drm_driver` static information fields, and will then describe individual operations in details as they get used in later sections.

2.1.1 Driver Information

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.

2.1.2 Module Initialization

This library provides helpers registering DRM drivers during module initialization and shut-down. The provided helpers act like bus-specific module helpers, such as `module_pci_driver()`, but respect additional parameters that control DRM driver registration.

Below is an example of initializing a DRM driver for a device on the PCI bus.

```
struct pci_driver my_pci_drv = {  
};  
  
drm_module_pci_driver(my_pci_drv);
```

The generated code will test if DRM drivers are enabled and register the PCI driver `my_pci_drv`. For more complex module initialization, you can still use `module_init()` and `module_exit()` in your driver.

2.1.3 Managing Ownership of the Framebuffer Aperture

A graphics device might be supported by different drivers, but only one driver can be active at any given time. Many systems load a generic graphics drivers, such as `EFI-GOP` or `VESA`, early during the boot process. During later boot stages, they replace the generic driver with a dedicated, hardware-specific driver. To take over the device the dedicated driver first has to remove the generic driver. DRM aperture functions manage ownership of DRM framebuffer memory and hand-over between drivers.

DRM drivers should call `drm_aperture_remove_conflicting_framebuffers()` at the top of their probe function. The function removes any generic driver that is currently associated with the given framebuffer memory. If the framebuffer is located at PCI BAR 0, the rsp code looks as in the example given below.

```
static const struct drm_driver example_driver = {  
    ...  
};  
  
static int remove_conflicting_framebuffers(struct pci_dev *pdev)  
{  
    resource_size_t base, size;  
    int ret;
```

```

    base = pci_resource_start(pdev, 0);
    size = pci_resource_len(pdev, 0);

    return drm_aperture_remove_conflicting_framebuffers(base, size,
                                                         &example_driver);
}

static int probe(struct pci_dev *pdev)
{
    int ret;

    // Remove any generic drivers...
    ret = remove_conflicting_framebuffers(pdev);
    if (ret)
        return ret;

    // ... and initialize the hardware.
    ...

    drm_dev_register();

    return 0;
}

```

PCI device drivers should call `drm_aperture_remove_conflicting_pci_framebuffers()` and let it detect the framebuffer apertures automatically. Device drivers without knowledge of the framebuffer's location shall call `drm_aperture_remove_framebuffers()`, which removes all drivers for known framebuffer.

Drivers that are susceptible to being removed by other drivers, such as generic EFI or VESA drivers, have to register themselves as owners of their given framebuffer memory. Ownership of the framebuffer memory is achieved by calling `devm_aperture_acquire_from_firmware()`. On success, the driver is the owner of the framebuffer range. The function fails if the framebuffer is already owned by another driver. See below for an example.

```

static int acquire_framebuffers(struct drm_device *dev, struct platform_device_
↪ *pdev)
{
    resource_size_t base, size;

    mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!mem)
        return -EINVAL;
    base = mem->start;
    size = resource_size(mem);

    return devm_acquire_aperture_from_firmware(dev, base, size);
}

static int probe(struct platform_device *pdev)

```

```
{
    struct drm_device *dev;
    int ret;

    // ... Initialize the device...
    dev = devm_drm_dev_alloc();
    ...

    // ... and acquire ownership of the framebuffer.
    ret = acquire_framebuffers(dev, pdev);
    if (ret)
        return ret;

    drm_dev_register(dev, 0);

    return 0;
}
```

The generic driver is now subject to forced removal by other drivers. This only works for platform drivers that support hot unplug. When a driver calls `drm_aperture_remove_conflicting_framebuffers()` et al. for the registered framebuffer range, the aperture helpers call `platform_device_unregister()` and the generic driver unloads itself. It may not access the device's registers, framebuffer memory, ROM, etc afterwards.

int **drm_aperture_remove_framebuffers**(const struct *drm_driver* *req_driver)
remove all existing framebuffers

Parameters

const struct drm_driver *req_driver
requesting DRM driver

Description

This function removes all graphics device drivers. Use this function on systems that can have their framebuffer located anywhere in memory.

Return

0 on success, or a negative errno code otherwise

int **devm_aperture_acquire_from_firmware**(struct *drm_device* *dev, resource_size_t base, resource_size_t size)

Acquires ownership of a firmware framebuffer on behalf of a DRM driver.

Parameters

struct drm_device *dev
the DRM device to own the framebuffer memory

resource_size_t base
the framebuffer's byte offset in physical memory

resource_size_t size
the framebuffer size in bytes

Description

Installs the given device as the new owner of the framebuffer. The function expects the framebuffer to be provided by a platform device that has been set up by firmware. Firmware can be any generic interface, such as EFI, VESA, VGA, etc. If the native hardware driver takes over ownership of the framebuffer range, the firmware state gets lost. Aperture helpers will then unregister the platform device automatically. Acquired apertures are released automatically if the underlying device goes away.

The function fails if the framebuffer range, or parts of it, is currently owned by another driver. To evict current owners, callers should use [`drm_aperture_remove_conflicting_framebuffers\(\)`](#) et al. before calling this function. The function also fails if the given device is not a platform device.

Return

0 on success, or a negative errno value otherwise.

int **drm_aperture_remove_conflicting_framebuffers**(resource_size_t base, resource_size_t size, const struct [drm_driver](#) *req_driver)

remove existing framebuffers in the given range

Parameters

resource_size_t base
the aperture's base address in physical memory

resource_size_t size
aperture size in bytes

const struct drm_driver *req_driver
requesting DRM driver

Description

This function removes graphics device drivers which use the memory range described by **base** and **size**.

Return

0 on success, or a negative errno code otherwise

int **drm_aperture_remove_conflicting_pci_framebuffers**(struct pci_dev *pdev, const struct [drm_driver](#) *req_driver)

remove existing framebuffers for PCI devices

Parameters

struct pci_dev *pdev
PCI device

const struct drm_driver *req_driver
requesting DRM driver

Description

This function removes graphics device drivers using the memory range configured for any of **pdev**'s memory bars. The function assumes that a PCI device with shadowed ROM drives a primary display and so kicks out vga16fb.

Return

0 on success, or a negative errno code otherwise

2.1.4 Device Instance and Driver Handling

A device instance for a drm driver is represented by `struct drm_device`. This is allocated and initialized with `devm_drm_dev_alloc()`, usually from bus-specific `->probe()` callbacks implemented by the driver. The driver then needs to initialize all the various subsystems for the drm device like memory management, vblank handling, modesetting support and initial output configuration plus obviously initialize all the corresponding hardware bits. Finally when everything is up and running and ready for userspace the device instance can be published using `drm_dev_register()`.

There is also deprecated support for initializing device instances using bus-specific helpers and the `drm_driver.load` callback. But due to backwards-compatibility needs the device instance have to be published too early, which requires unpretty global locking to make safe and is therefore only support for existing drivers not yet converted to the new scheme.

When cleaning up a device instance everything needs to be done in reverse: First unpublish the device instance with `drm_dev_unregister()`. Then clean up any other resources allocated at device initialization and drop the driver's reference to `drm_device` using `drm_dev_put()`.

Note that any allocation or resource which is visible to userspace must be released only when the final `drm_dev_put()` is called, and not when the driver is unbound from the underlying physical struct device. Best to use `drm_device` managed resources with `drmm_add_action()`, `drmm_kmalloc()` and related functions.

devres managed resources like `devm_kmalloc()` can only be used for resources directly related to the underlying hardware device, and only used in code paths fully protected by `drm_dev_enter()` and `drm_dev_exit()`.

Display driver example

The following example shows a typical structure of a DRM display driver. The example focus on the `probe()` function and the other functions that is almost always present and serves as a demonstration of `devm_drm_dev_alloc()`.

```
struct driver_device {
    struct drm_device drm;
    void *userspace_facing;
    struct clk *pclk;
};

static const struct drm_driver driver_drm_driver = {
    [...]
};

static int driver_probe(struct platform_device *pdev)
{
    struct driver_device *priv;
    struct drm_device *drm;
    int ret;
```



```

    priv = devm_drm_dev_alloc(&pdev->dev, &driver_drm_driver,
                             struct driver_device, drm);
    if (IS_ERR(priv))
        return PTR_ERR(priv);
    drm = &priv->drm;

    ret = drmm_mode_config_init(drm);
    if (ret)
        return ret;

    priv->userspace_facing = drmm_kzalloc(..., GFP_KERNEL);
    if (!priv->userspace_facing)
        return -ENOMEM;

    priv->pclk = devm_clk_get(dev, "PCLK");
    if (IS_ERR(priv->pclk))
        return PTR_ERR(priv->pclk);

    // Further setup, display pipeline etc

    platform_set_drvdata(pdev, drm);

    drm_mode_config_reset(drm);

    ret = drm_dev_register(drm);
    if (ret)
        return ret;

    drm_fbdev_generic_setup(drm, 32);

    return 0;
}

// This function is called before the devm_resources are released
static int driver_remove(struct platform_device *pdev)
{
    struct drm_device *drm = platform_get_drvdata(pdev);

    drm_dev_unregister(drm);
    drm_atomic_helper_shutdown(drm)

    return 0;
}

// This function is called on kernel restart and shutdown
static void driver_shutdown(struct platform_device *pdev)
{
    drm_atomic_helper_shutdown(platform_get_drvdata(pdev));
}

```

```
static int __maybe_unused driver_pm_suspend(struct device *dev)
{
    return drm_mode_config_helper_suspend(dev_get_drvdata(dev));
}

static int __maybe_unused driver_pm_resume(struct device *dev)
{
    drm_mode_config_helper_resume(dev_get_drvdata(dev));

    return 0;
}

static const struct dev_pm_ops driver_pm_ops = {
    SET_SYSTEM_SLEEP_PM_OPS(driver_pm_suspend, driver_pm_resume)
};

static struct platform_driver driver_driver = {
    .driver = {
        [...]
        .pm = &driver_pm_ops,
    },
    .probe = driver_probe,
    .remove = driver_remove,
    .shutdown = driver_shutdown,
};
module_platform_driver(driver_driver);
```

Drivers that want to support device unplugging (USB, DT overlay unload) should use *drm_dev_unplug()* instead of *drm_dev_unregister()*. The driver must protect regions that is accessing device resources to prevent use after they're released. This is done using *drm_dev_enter()* and *drm_dev_exit()*. There is one shortcoming however, *drm_dev_unplug()* marks the *drm_device* as unplugged before *drm_atomic_helper_shutdown()* is called. This means that if the disable code paths are protected, they will not run on regular driver module unload, possibly leaving the hardware enabled.

enum **switch_power_state**
power state of drm device

Constants

DRM_SWITCH_POWER_ON
Power state is ON

DRM_SWITCH_POWER_OFF
Power state is OFF

DRM_SWITCH_POWER_CHANGING
Power state is changing

DRM_SWITCH_POWER_DYNAMIC_OFF
Suspended

struct **drm_device**
DRM device structure

Definition:

```

struct drm_device {
    int if_version;
    struct kref ref;
    struct device *dev;
    struct {
        struct list_head resources;
        void *final_kfree;
        spinlock_t lock;
    } managed;
    const struct drm_driver *driver;
    void *dev_private;
    struct drm_minor *primary;
    struct drm_minor *render;
    struct drm_minor *accel;
    bool registered;
    struct drm_master *master;
    u32 driver_features;
    bool unplugged;
    struct inode *anon_inode;
    char *unique;
    struct mutex struct_mutex;
    struct mutex master_mutex;
    atomic_t open_count;
    struct mutex filelist_mutex;
    struct list_head filelist;
    struct list_head filelist_internal;
    struct mutex clientlist_mutex;
    struct list_head clientlist;
    bool vblank_disable_immediate;
    struct drm_vblank_crtc *vblank;
    spinlock_t vblank_time_lock;
    spinlock_t vbl_lock;
    u32 max_vblank_count;
    struct list_head vblank_event_list;
    spinlock_t event_lock;
    unsigned int num_crtcs;
    struct drm_mode_config mode_config;
    struct mutex object_name_lock;
    struct idr object_name_idr;
    struct drm_vma_offset_manager *vma_offset_manager;
    struct drm_vram_mm *vram_mm;
    enum switch_power_state switch_power_state;
    struct drm_fb_helper *fb_helper;
    struct mutex debugfs_mutex;
    struct list_head debugfs_list;
};

```

Members**if_version**

Highest interface version set

ref

Object ref-count

dev

Device structure of bus-device

managed

Managed resources linked to the lifetime of this *drm_device* as tracked by **ref**.

driver

DRM driver managing the device

dev_private

DRM driver private data. This is deprecated and should be left set to NULL.

Instead of using this pointer it is recommended that drivers use *devm_drm_dev_alloc()* and embed struct *drm_device* in their larger per-device structure.

primary

Primary node. Drivers should not interact with this directly. debugfs interfaces can be registered with *drm_debugfs_add_file()*, and sysfs should be directly added on the hardware (and not character device node) struct device **dev**.

render

Render node. Drivers should not interact with this directly ever. Drivers should not expose any additional interfaces in debugfs or sysfs on this node.

accel

Compute Acceleration node

registered

Internally used by *drm_dev_register()* and *drm_connector_register()*.

master

Currently active master for this device. Protected by *master_mutex*

driver_features

per-device driver features

Drivers can clear specific flags here to disallow certain features on a per-device basis while still sharing a single *struct drm_driver* instance across all devices.

unplugged

Flag to tell if the device has been unplugged. See *drm_dev_enter()* and *drm_dev_is_unplugged()*.

anon_inode

inode for private address-space

unique

Unique name of the device

struct_mutex

Lock for others (not *drm_minor.master* and *drm_file.is_master*)

WARNING: Only drivers annotated with DRIVER_LEGACY should be using this.

master_mutex

Lock for *drm_minor.master* and *drm_file.is_master*

open_count

Usage counter for outstanding files open, protected by `drm_global_mutex`

filelist_mutex

Protects **filelist**.

filelist

List of userspace clients, linked through `drm_file.lhead`.

filelist_internal

List of open DRM files for in-kernel clients. Protected by `filelist_mutex`.

clientlist_mutex

Protects `clientlist` access.

clientlist

List of in-kernel clients. Protected by `clientlist_mutex`.

vblank_disable_immediate

If true, vblank interrupt will be disabled immediately when the refcount drops to zero, as opposed to via the vblank disable timer.

This can be set to true if the hardware has a working vblank counter with high-precision timestamping (otherwise there are races) and the driver uses `drm_crtc_vblank_on()` and `drm_crtc_vblank_off()` appropriately. See also **max_vblank_count** and `drm_crtc_funcs.get_vblank_counter`.

vblank

Array of vblank tracking structures, one per `struct drm_crtc`. For historical reasons (vblank support predates kernel modesetting) this is free-standing and not part of `struct drm_crtc` itself. It must be initialized explicitly by calling `drm_vblank_init()`.

vblank_time_lock

Protects vblank count and time updates during vblank enable/disable

vbl_lock

Top-level vblank references lock, wraps the low-level **vblank_time_lock**.

max_vblank_count

Maximum value of the vblank registers. This value +1 will result in a wrap-around of the vblank register. It is used by the vblank core to handle wrap-arounds.

If set to zero the vblank core will try to guess the elapsed vblanks between times when the vblank interrupt is disabled through high-precision timestamps. That approach is suffering from small races and imprecision over longer time periods, hence exposing a hardware vblank counter is always recommended.

This is the statically configured device wide maximum. The driver can instead choose to use a runtime configurable per-crtc value `drm_vblank_crtc.max_vblank_count`, in which case **max_vblank_count** must be left at zero. See `drm_crtc_set_max_vblank_count()` on how to use the per-crtc value.

If non-zero, `drm_crtc_funcs.get_vblank_counter` must be set.

vblank_event_list

List of vblank events

event_lock

Protects **vblank_event_list** and event delivery in general. See `drm_send_event()` and

drm_send_event_locked().

num_crtcs

Number of CRTC's on this device

mode_config

Current mode config

object_name_lock

GEM information

object_name_idr

GEM information

vma_offset_manager

GEM information

vram_mm

VRAM MM memory manager

switch_power_state

Power state of the client. Used by drivers supporting the switcheroo driver. The state is maintained in the *vga_switcheroo_client_ops.set_gpu_state* callback

fb_helper

Pointer to the fbdev emulation structure. Set by *drm_fb_helper_init()* and cleared by *drm_fb_helper_fini()*.

debugfs_mutex

Protects debugfs_list access.

debugfs_list

List of debugfs files to be created by the DRM device. The files must be added during *drm_dev_register()*.

Description

This structure represent a complete card that may contain multiple heads.

enum **drm_driver_feature**

feature flags

Constants

DRIVER_GEM

Driver use the GEM memory manager. This should be set for all modern drivers.

DRIVER_MODESET

Driver supports mode setting interfaces (KMS).

DRIVER_RENDER

Driver supports dedicated render nodes. See also the *section on render nodes* for details.

DRIVER_ATOMIC

Driver supports the full atomic modesetting userspace API. Drivers which only use atomic internally, but do not support the full userspace API (e.g. not all properties converted to atomic, or multi-plane updates are not guaranteed to be tear-free) should not set this flag.

DRIVER_SYNCOBJ

Driver supports *drm_syncobj* for explicit synchronization of command submission.

DRIVER_SYNCOBJ_TIMELINE

Driver supports the timeline flavor of *drm_syncobj* for explicit synchronization of command submission.

DRIVER_COMPUTE_ACCEL

Driver supports compute acceleration devices. This flag is mutually exclusive with **DRIVER_RENDER** and **DRIVER_MODESET**. Devices that support both graphics and compute acceleration should be handled by two drivers that are connected using auxiliary bus.

DRIVER_GEM_GPUVA

Driver supports user defined GPU VA bindings for GEM objects.

DRIVER_CURSOR_HOTSPOT

Driver supports and requires cursor hotspot information in the cursor plane (e.g. cursor plane has to actually track the mouse cursor and the clients are required to set hotspot in order for the cursor planes to work correctly).

DRIVER_USE_AGP

Set up DRM AGP support, see *drm_agp_init()*, the DRM core will manage AGP resources. New drivers don't need this.

DRIVER_LEGACY

Denote a legacy driver using shadow attach. Do not use.

DRIVER_PCI_DMA

Driver is capable of PCI DMA, mapping of PCI DMA buffers to userspace will be enabled. Only for legacy drivers. Do not use.

DRIVER_SG

Driver can perform scatter/gather DMA, allocation and mapping of scatter/gather buffers will be enabled. Only for legacy drivers. Do not use.

DRIVER_HAVE_DMA

Driver supports DMA, the userspace DMA API will be supported. Only for legacy drivers. Do not use.

DRIVER_HAVE_IRQ

Legacy irq support. Only for legacy drivers. Do not use.

Description

See *drm_driver.driver_features*, *drm_device.driver_features* and *drm_core_check_feature()*.

struct *drm_driver*

DRM driver structure

Definition:

```
struct drm_driver {
    int (*load) (struct drm_device *, unsigned long flags);
    int (*open) (struct drm_device *, struct drm_file *);
    void (*postclose) (struct drm_device *, struct drm_file *);
    void (*lastclose) (struct drm_device *);
    void (*unload) (struct drm_device *);
    void (*release) (struct drm_device *);
    void (*master_set)(struct drm_device *dev, struct drm_file *file_priv,
```

```

→bool from_open);
    void (*master_drop)(struct drm_device *dev, struct drm_file *file_priv);
    void (*debugfs_init)(struct drm_minor *minor);
    struct drm_gem_object *(*gem_create_object)(struct drm_device *dev, size_t,
→size);
    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);
    struct drm_gem_object * (*gem_prime_import)(struct drm_device *dev, struct
→dma_buf *dma_buf);
    struct drm_gem_object *(*gem_prime_import_sg_table)(struct drm_device *dev,
→struct dma_buf_attachment *attach, struct sg_table *sgt);
    int (*dumb_create)(struct drm_file *file_priv, struct drm_device *dev,
→struct drm_mode_create_dumb *args);
    int (*dumb_map_offset)(struct drm_file *file_priv, struct drm_device *dev,
→uint32_t handle, uint64_t *offset);
    void (*show_fdinfo)(struct drm_printer *p, struct drm_file *f);
    int major;
    int minor;
    int patchlevel;
    char *name;
    char *desc;
    char *date;
    u32 driver_features;
    const struct drm_ioctl_desc *ioctls;
    int num_ioctls;
    const struct file_operations *fops;
#ifdef CONFIG_DRM_LEGACY;
};

```

Members

load

Backward-compatible driver callback to complete initialization steps after the driver is registered. For this reason, may suffer from race conditions and its use is deprecated for new drivers. It is therefore only supported for existing drivers not yet converted to the new scheme. See `devm_drm_dev_alloc()` and `drm_dev_register()` for proper and race-free way to set up a `struct drm_device`.

This is deprecated, do not use!

Returns:

Zero on success, non-zero value on failure.

open

Driver callback when a new `struct drm_file` is opened. Useful for setting up driver-private data structures like buffer allocators, execution contexts or similar things. Such driver-private resources must be released again in **postclose**.

Since the display/modetest side of DRM can only be owned by exactly one `struct drm_file` (see `drm_file.is_master` and `drm_device.master`) there should never be a need to set up any modeset related resources in this callback. Doing so would be a driver design bug.

Returns:

0 on success, a negative error code on failure, which will be promoted to userspace as the result of the `open()` system call.

postclose

One of the driver callbacks when a new `struct drm_file` is closed. Useful for tearing down driver-private data structures allocated in **open** like buffer allocators, execution contexts or similar things.

Since the display/modeset side of DRM can only be owned by exactly one `struct drm_file` (see `drm_file.is_master` and `drm_device.master`) there should never be a need to tear down any modeset related resources in this callback. Doing so would be a driver design bug.

lastclose

Called when the last `struct drm_file` has been closed and there's currently no userspace client for the `struct drm_device`.

Modern drivers should only use this to force-restore the fbdev framebuffer using `drm_fb_helper_restore_fbdev_mode_unlocked()`. Anything else would indicate there's something seriously wrong. Modern drivers can also use this to execute delayed power switching state changes, e.g. in conjunction with the *VGA Switcheroo* infrastructure.

This is called after **postclose** hook has been called.

NOTE:

All legacy drivers use this callback to de-initialize the hardware. This is purely because of the shadow-attach model, where the DRM kernel driver does not really own the hardware. Instead ownership is handled with the help of userspace through an inheritedly racy dance to set/unset the VT into raw mode.

Legacy drivers initialize the hardware in the **firstopen** callback, which isn't even called for modern drivers.

unload

Reverse the effects of the driver load callback. Ideally, the clean up performed by the driver should happen in the reverse order of the initialization. Similarly to the load hook, this handler is deprecated and its usage should be dropped in favor of an open-coded teardown function at the driver layer. See `drm_dev_unregister()` and `drm_dev_put()` for the proper way to remove a `struct drm_device`.

The `unload()` hook is called right after unregistering the device.

release

Optional callback for destroying device data after the final reference is released, i.e. the device is being destroyed.

This is deprecated, clean up all memory allocations associated with a `drm_device` using `drm_mm_add_action()`, `drm_mm_kmalloc()` and related managed resources functions.

master_set

Called whenever the minor master is set. Only used by vmwgfx.

master_drop

Called whenever the minor master is dropped. Only used by vmwgfx.

debugfs_init

Allows drivers to create driver-specific debugfs files.

gem_create_object

constructor for gem objects

Hook for allocating the GEM object struct, for use by the CMA and SHMEM GEM helpers. Returns a GEM object on success, or an ERR_PTR()-encoded error code otherwise.

prime_handle_to_fd

PRIME export function. Only used by vmwgfx.

prime_fd_to_handle

PRIME import function. Only used by vmwgfx.

gem_prime_import

Import hook for GEM drivers.

This defaults to `drm_gem_prime_import()` if not set.

gem_prime_import_sg_table

Optional hook used by the PRIME helper functions `drm_gem_prime_import()` respectively `drm_gem_prime_import_dev()`.

dumb_create

This creates a new dumb buffer in the driver's backing storage manager (GEM, TTM or something else entirely) and returns the resulting buffer handle. This handle can then be wrapped up into a framebuffer modeset object.

Note that userspace is not allowed to use such objects for render acceleration - drivers must create their own private ioctls for such a use case.

Width, height and depth are specified in the `drm_mode_create_dumb` argument. The callback needs to fill the handle, pitch and size for the created buffer.

Called by the user via ioctl.

Returns:

Zero on success, negative errno on failure.

dumb_map_offset

Allocate an offset in the drm device node's address space to be able to memory map a dumb buffer.

The default implementation is `drm_gem_create_mmap_offset()`. GEM based drivers must not overwrite this.

Called by the user via ioctl.

Returns:

Zero on success, negative errno on failure.

show_fdinfo

Print device specific fdinfo. See *DRM client usage stats*.

major

driver major number

minor

driver minor number

patchlevel

driver patch level

name

driver name

desc

driver description

date

driver date

driver_features

Driver features, see [enum `drm_driver_feature`](#). Drivers can disable some features on a per-instance basis using [`drm_device.driver_features`](#).

ioctls

Array of driver-private IOCTL description entries. See the chapter on [IOCTL support in the *userland interfaces* chapter](#) for the full details.

num_ioctls

Number of entries in **ioctls**.

fops

File operations for the DRM device node. See the discussion in [file operations](#) for in-depth coverage and some examples.

Description

This structure represent the common code for a family of cards. There will be one [struct `drm_device`](#) for each card present in this family. It contains lots of vfunc entries, and a pile of those probably should be moved to more appropriate places like [`drm_mode_config_funcs`](#) or into a new operations structure for GEM drivers.

devm_drm_dev_alloc

`devm_drm_dev_alloc (parent, driver, type, member)`

Resource managed allocation of a [`drm_device`](#) instance

Parameters**parent**

Parent device object

driver

DRM driver

type

the type of the struct which contains struct [`drm_device`](#)

member

the name of the [`drm_device`](#) within **type**.

Description

This allocates and initialize a new DRM device. No device registration is done. Call [`drm_dev_register\(\)`](#) to advertice the device to user space and register it with other core sub-systems. This should be done last in the device initialization sequence to make sure userspace can't access an inconsistent state.

The initial ref-count of the object is 1. Use *drm_dev_get()* and *drm_dev_put()* to take and drop further ref-counts.

It is recommended that drivers embed *struct drm_device* into their own device structure.

Note that this manages the lifetime of the resulting *drm_device* automatically using devres. The DRM device initialized with this function is automatically put on driver detach using *drm_dev_put()*.

Return

Pointer to new DRM device, or ERR_PTR on failure.

bool **drm_dev_is_unplugged**(struct *drm_device* *dev)
is a DRM device unplugged

Parameters

struct drm_device *dev
DRM device

Description

This function can be called to check whether a hotpluggable is unplugged. Unplugging itself is signalled through *drm_dev_unplug()*. If a device is unplugged, these two functions guarantee that any store before calling *drm_dev_unplug()* is visible to callers of this function after it completes

WARNING: This function fundamentally races against *drm_dev_unplug()*. It is recommended that drivers instead use the underlying *drm_dev_enter()* and *drm_dev_exit()* function pairs.

bool **drm_core_check_all_features**(const struct *drm_device* *dev, u32 features)
check driver feature flags mask

Parameters

const struct drm_device *dev
DRM device to check

u32 features
feature flag(s) mask

Description

This checks **dev** for driver features, see *drm_driver.driver_features*, *drm_device.driver_features*, and the various *enum drm_driver_feature* flags.

Returns true if all features in the **features** mask are supported, false otherwise.

bool **drm_core_check_feature**(const struct *drm_device* *dev, enum *drm_driver_feature* feature)
check driver feature flags

Parameters

const struct drm_device *dev
DRM device to check

enum drm_driver_feature feature
feature flag

Description

This checks **dev** for driver features, see [drm_driver.driver_features](#), [drm_device.driver_features](#), and the various [enum drm_driver_feature](#) flags.

Returns true if the **feature** is supported, false otherwise.

bool **drm_drv_uses_atomic_modeset**(struct [drm_device](#) *dev)
check if the driver implements atomic_commit()

Parameters

struct drm_device *dev
DRM device

Description

This check is useful if drivers do not have DRIVER_ATOMIC set but have atomic modesetting internally implemented.

void **drm_put_dev**(struct [drm_device](#) *dev)
Unregister and release a DRM device

Parameters

struct drm_device *dev
DRM device

Description

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

Cleans up all DRM device, calling [drm_lastclose\(\)](#).

Note

Use of this function is deprecated. It will eventually go away completely. Please use [drm_dev_unregister\(\)](#) and [drm_dev_put\(\)](#) explicitly instead to make sure that the device isn't userspace accessible any more while teardown is in progress, ensuring that userspace can't access an inconsistent state.

bool **drm_dev_enter**(struct [drm_device](#) *dev, int *idx)
Enter device critical section

Parameters

struct drm_device *dev
DRM device

int *idx
Pointer to index that will be passed to the matching [drm_dev_exit\(\)](#)

Description

This function marks and protects the beginning of a section that should not be entered after the device has been unplugged. The section end is marked with [drm_dev_exit\(\)](#). Calls to this function can be nested.

Return

True if it is OK to enter the section, false otherwise.

void **drm_dev_exit**(int idx)

Exit device critical section

Parameters

int **idx**

index returned from [drm_dev_enter\(\)](#)

Description

This function marks the end of a section that should not be entered after the device has been unplugged.

void **drm_dev_unplug**(struct [drm_device](#) *dev)

unplug a DRM device

Parameters

struct [drm_device](#) *dev

DRM device

Description

This unplugs a hotpluggable DRM device, which makes it inaccessible to userspace operations. Entry-points can use [drm_dev_enter\(\)](#) and [drm_dev_exit\(\)](#) to protect device resources in a race free manner. This essentially unregisters the device like [drm_dev_unregister\(\)](#), but can be called while there are still open users of **dev**.

struct [drm_device](#) ***drm_dev_alloc**(const struct [drm_driver](#) *driver, struct device *parent)

Allocate new DRM device

Parameters

const struct [drm_driver](#) *driver

DRM driver to allocate device for

struct device *parent

Parent device object

Description

This is the deprecated version of `devm_drm_dev_alloc()`, which does not support subclassing through embedding the struct [drm_device](#) in a driver private structure, and which does not support automatic cleanup through devres.

Return

Pointer to new DRM device, or ERR_PTR on failure.

void **drm_dev_get**(struct [drm_device](#) *dev)

Take reference of a DRM device

Parameters

struct [drm_device](#) *dev

device to take reference of or NULL

Description

This increases the ref-count of **dev** by one. You *must* already own a reference when calling this. Use [drm_dev_put\(\)](#) to drop this reference again.

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

```
void drm_dev_put(struct drm_device *dev)
```

Drop reference of a DRM device

Parameters

```
struct drm_device *dev
```

device to drop reference of or NULL

Description

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

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

Register DRM device

Parameters

```
struct drm_device *dev
```

Device to register

```
unsigned long flags
```

Flags passed to the driver's .load() function

Description

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

Never call this twice on any device!

NOTE

To ensure backward compatibility with existing drivers method this function calls the *drm_driver.load* method after registering the device nodes, creating race conditions. Usage of the *drm_driver.load* methods is therefore deprecated, drivers must perform all initialization before calling *drm_dev_register()*.

Return

0 on success, negative error code on failure.

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

Unregister DRM device

Parameters

```
struct drm_device *dev
```

Device to unregister

Description

Unregister the DRM device from the system. This does the reverse of *drm_dev_register()* but does not deallocate the device. The caller must call *drm_dev_put()* to drop their final reference, unless it is managed with devres (as devices allocated with `devm_drm_dev_alloc()` are), in which case there is already an unwind action registered.

A special form of unregistering for hotpluggable devices is `drm_dev_unplug()`, which can be called while there are still open users of `dev`.

This should be called first in the device teardown code to make sure userspace can't access the device instance any more.

2.1.5 Driver Load

Component Helper Usage

DRM drivers that drive hardware where a logical device consists of a pile of independent hardware blocks are recommended to use the component helper library. For consistency and better options for code reuse the following guidelines apply:

- The entire device initialization procedure should be run from the `component_master_ops.master_bind` callback, starting with `devm_drm_dev_alloc()`, then binding all components with `component_bind_all()` and finishing with `drm_dev_register()`.
- The opaque pointer passed to all components through `component_bind_all()` should point at `struct drm_device` of the device instance, not some driver specific private structure.
- The component helper fills the niche where further standardization of interfaces is not practical. When there already is, or will be, a standardized interface like `drm_bridge` or `drm_panel`, providing its own functions to find such components at driver load time, like `drm_of_find_panel_or_bridge()`, then the component helper should not be used.

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 ? 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.

2.1.6 Managed Resources

Inspired by struct device managed resources, but tied to the lifetime of struct *drm_device*, which can outlive the underlying physical device, usually when userspace has some open files and other handles to resources still open.

Release actions can be added with *drmm_add_action()*, memory allocations can be done directly with *drmm_kmalloc()* and the related functions. Everything will be released on the final *drm_dev_put()* in reverse order of how the release actions have been added and memory has been allocated since driver loading started with *devm_drm_dev_alloc()*.

Note that release actions and managed memory can also be added and removed during the lifetime of the driver, all the functions are fully concurrent safe. But it is recommended to use managed resources only for resources that change rarely, if ever, during the lifetime of the *drm_device* instance.

```
void *drmm_kmalloc(struct drm_device *dev, size_t size, gfp_t gfp)
    drm_device managed kmalloc()
```

Parameters

struct *drm_device* *dev
DRM device

size_t size
size of the memory allocation

gfp_t gfp
GFP allocation flags

Description

This is a *drm_device* managed version of *kmalloc()*. The allocated memory is automatically freed on the final *drm_dev_put()*. Memory can also be freed before the final *drm_dev_put()* by calling *drmm_kfree()*.

```
char *drmm_kstrdup(struct drm_device *dev, const char *s, gfp_t gfp)
    drm_device managed kstrdup()
```

Parameters

struct *drm_device* *dev
DRM device

const char *s
0-terminated string to be duplicated

gfp_t gfp
GFP allocation flags

Description

This is a *drm_device* managed version of *kstrdup()*. The allocated memory is automatically freed on the final *drm_dev_put()* and works exactly like a memory allocation obtained by *drmm_kmalloc()*.

```
void drmm_kfree(struct drm_device *dev, void *data)
    drm_device managed kfree()
```

Parameters

struct drm_device *dev

DRM device

void *data

memory allocation to be freed

Description

This is a *drm_device* managed version of `kfree()` which can be used to release memory allocated through *drmm_kmalloc()* or any of its related functions before the final *drm_dev_put()* of **dev**.

drmm_add_action

`drmm_add_action (dev, action, data)`

add a managed release action to a *drm_device*

Parameters

dev

DRM device

action

function which should be called when **dev** is released

data

opaque pointer, passed to **action**

Description

This function adds the **release** action with optional parameter **data** to the list of cleanup actions for **dev**. The cleanup actions will be run in reverse order in the final *drm_dev_put()* call for **dev**.

drmm_add_action_or_reset

`drmm_add_action_or_reset (dev, action, data)`

add a managed release action to a *drm_device*

Parameters

dev

DRM device

action

function which should be called when **dev** is released

data

opaque pointer, passed to **action**

Description

Similar to *drmm_add_action()*, with the only difference that upon failure **action** is directly called for any cleanup work necessary on failures.

`void *drmm_kzalloc(struct drm_device *dev, size_t size, gfp_t gfp)`

drm_device managed `kzalloc()`

Parameters

struct drm_device *dev

DRM device

size_t size

size of the memory allocation

gfp_t gfp

GFP allocation flags

Description

This is a *drm_device* managed version of `kzalloc()`. The allocated memory is automatically freed on the final *drm_dev_put()*. Memory can also be freed before the final *drm_dev_put()* by calling *drmm_kfree()*.

`void *drmm_kmalloc_array(struct drm_device *dev, size_t n, size_t size, gfp_t flags)`
drm_device managed `kmalloc_array()`

Parameters

struct *drm_device* *dev

DRM device

size_t n

number of array elements to allocate

size_t size

size of array member

gfp_t flags

GFP allocation flags

Description

This is a *drm_device* managed version of `kmalloc_array()`. The allocated memory is automatically freed on the final *drm_dev_put()* and works exactly like a memory allocation obtained by *drmm_kmalloc()*.

`void *drmm_kcalloc(struct drm_device *dev, size_t n, size_t size, gfp_t flags)`
drm_device managed `kcalloc()`

Parameters

struct *drm_device* *dev

DRM device

size_t n

number of array elements to allocate

size_t size

size of array member

gfp_t flags

GFP allocation flags

Description

This is a *drm_device* managed version of `kcalloc()`. The allocated memory is automatically freed on the final *drm_dev_put()* and works exactly like a memory allocation obtained by *drmm_kmalloc()*.

drmm_mutex_init

`drmm_mutex_init (dev, lock)`

drm_device-managed `mutex_init()`

Parameters

dev

DRM device

lock

lock to be initialized

Return

0 on success, or a negative errno code otherwise.

Description

This is a *drm_device*-managed version of `mutex_init()`. The initialized lock is automatically destroyed on the final *drm_dev_put()*.

2.1.7 Bus-specific Device Registration and PCI Support

A number of functions are provided to help with device registration. The functions deal with PCI and platform devices respectively and are only provided for historical reasons. These are all deprecated and shouldn't be used in new drivers. Besides that there's a few helpers for pci drivers.

`int drm_legacy_pci_init(const struct drm_driver *driver, struct pci_driver *pdriver)`
shadow-attach a legacy DRM PCI driver

Parameters

`const struct drm_driver *driver`
DRM device driver

`struct pci_driver *pdriver`
PCI device driver

Description

This is only used by legacy dri1 drivers and deprecated.

Return

0 on success or a negative error code on failure.

`void drm_legacy_pci_exit(const struct drm_driver *driver, struct pci_driver *pdriver)`
unregister shadow-attach legacy DRM driver

Parameters

`const struct drm_driver *driver`
DRM device driver

`struct pci_driver *pdriver`
PCI device driver

Description

Unregister a DRM driver shadow-attached through *drm_legacy_pci_init()*. This is deprecated and only used by dri1 drivers.

2.2 Open/Close, File Operations and IOCTLs

2.2.1 File Operations

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 resulting struct `file_operations` must be stored in the `drm_driver.fops` field. The mandatory functions are `drm_open()`, `drm_read()`, `drm_ioctl()` and `drm_compat_ioctl()` if `CONFIG_COMPAT` is enabled. Note that `drm_compat_ioctl` will be `NULL` if `CONFIG_COMPAT=n`, so there's no need to sprinkle `#ifdef` into the code. Drivers which implement private ioctls that require 32/64 bit compatibility support must provide their own `file_operations.compat_ioctl` handler that processes private ioctls and calls `drm_compat_ioctl()` for core ioctls.

In addition `drm_read()` and `drm_poll()` provide support for DRM events. DRM events are a generic and extensible means to send asynchronous events to userspace through the file descriptor. They are used to send vblank event and page flip completions by the KMS API. But drivers can also use it for their own needs, e.g. to signal completion of rendering.

For the driver-side event interface see `drm_event_reserve_init()` and `drm_send_event()` as the main starting points.

The memory mapping implementation will vary depending on how the driver manages memory. Legacy drivers will use the deprecated `drm_legacy_mmap()` function, modern drivers should use one of the provided memory-manager specific implementations. For GEM-based drivers this is `drm_gem_mmap()`.

No other file operations are supported by the DRM userspace API. Overall the following is an example `file_operations` structure:

```
static const example_drm_fops = {
    .owner = THIS_MODULE,
    .open = drm_open,
    .release = drm_release,
    .unlocked_ioctl = drm_ioctl,
    .compat_ioctl = drm_compat_ioctl, // NULL if CONFIG_COMPAT=n
    .poll = drm_poll,
    .read = drm_read,
    .llseek = no_llseek,
    .mmap = drm_gem_mmap,
};
```

For plain GEM based drivers there is the `DEFINE_DRM_GEM_FOPS()` macro, and for DMA based drivers there is the `DEFINE_DRM_GEM_DMA_FOPS()` macro to make this simpler.

The driver's `file_operations` must be stored in `drm_driver.fops`.

For driver-private IOCTL handling see the more detailed discussion in *IOCTL support in the userland interfaces chapter*.

struct **drm_minor**

DRM device minor structure

Definition:

```
struct drm_minor {  
};
```

Members

Description

This structure represents a DRM minor number for device nodes in /dev. Entirely opaque to drivers and should never be inspected directly by drivers. Drivers instead should only interact with *struct drm_file* and of course *struct drm_device*, which is also where driver-private data and resources can be attached to.

struct **drm_pending_event**

Event queued up for userspace to read

Definition:

```
struct drm_pending_event {  
    struct completion *completion;  
    void (*completion_release)(struct completion *completion);  
    struct drm_event *event;  
    struct dma_fence *fence;  
    struct drm_file *file_priv;  
    struct list_head link;  
    struct list_head pending_link;  
};
```

Members

completion

Optional pointer to a kernel internal completion signalled when *drm_send_event()* is called, useful to internally synchronize with nonblocking operations.

completion_release

Optional callback currently only used by the atomic modeset helpers to clean up the reference count for the structure **completion** is stored in.

event

Pointer to the actual event that should be sent to userspace to be read using *drm_read()*. Can be optional, since nowadays events are also used to signal kernel internal threads with **completion** or DMA transactions using **fence**.

fence

Optional DMA fence to unblock other hardware transactions which depend upon the non-blocking DRM operation this event represents.

file_priv

struct drm_file where **event** should be delivered to. Only set when **event** is set.

link

Double-linked list to keep track of this event. Can be used by the driver up to the point when it calls *drm_send_event()*, after that this list entry is owned by the core for its own book-keeping.

pending_link

Entry on *drm_file.pending_event_list*, to keep track of all pending events for **file_priv**,

to allow correct unwinding of them when userspace closes the file before the event is delivered.

Description

This represents a DRM event. Drivers can use this as a generic completion mechanism, which supports kernel-internal struct completion, struct dma_fence and also the DRM-specific *struct drm_event* delivery mechanism.

struct **drm_file**

DRM file private data

Definition:

```
struct drm_file {
    bool authenticated;
    bool stereo_allowed;
    bool universal_planes;
    bool atomic;
    bool aspect_ratio_allowed;
    bool writeback_connectors;
    bool was_master;
    bool is_master;
    bool supports_virtualized_cursor_plane;
    struct drm_master *master;
    spinlock_t master_lookup_lock;
    struct pid __rcu *pid;
    u64 client_id;
    drm_magic_t magic;
    struct list_head lhead;
    struct drm_minor *minor;
    struct idr object_idr;
    spinlock_t table_lock;
    struct idr syncobj_idr;
    spinlock_t syncobj_table_lock;
    struct file *filp;
    void *driver_priv;
    struct list_head fbs;
    struct mutex fbs_lock;
    struct list_head blobs;
    wait_queue_head_t event_wait;
    struct list_head pending_event_list;
    struct list_head event_list;
    int event_space;
    struct mutex event_read_lock;
    struct drm_prime_file_private prime;
};
```

Members

authenticated

Whether the client is allowed to submit rendering, which for legacy nodes means it must be authenticated.

See also the [section on primary nodes and authentication](#).

stereo_allowed

True when the client has asked us to expose stereo 3D mode flags.

universal_planes

True if client understands CRTC primary planes and cursor planes in the plane list. Automatically set when **atomic** is set.

atomic

True if client understands atomic properties.

aspect_ratio_allowed

True, if client can handle picture aspect ratios, and has requested to pass this information along with the mode.

writeback_connectors

True if client understands writeback connectors

was_master

This client has or had, master capability. Protected by struct `drm_device.master_mutex`.

This is used to ensure that CAP_SYS_ADMIN is not enforced, if the client is or was master in the past.

is_master

This client is the creator of **master**. Protected by struct `drm_device.master_mutex`.

See also the [section on primary nodes and authentication](#).

supports_virtualized_cursor_plane

This client is capable of handling the cursor plane with the restrictions imposed on it by the virtualized drivers.

This implies that the cursor plane has to behave like a cursor i.e. track cursor movement. It also requires setting of the hotspot properties by the client on the cursor plane.

master

Master this node is currently associated with. Protected by struct `drm_device.master_mutex`, and serialized by **master_lookup_lock**.

Only relevant if `drm_is_primary_client()` returns true. Note that this only matches `drm_device.master` if the master is the currently active one.

To update **master**, both `drm_device.master_mutex` and **master_lookup_lock** need to be held, therefore holding either of them is safe and enough for the read side.

When dereferencing this pointer, either hold struct `drm_device.master_mutex` for the duration of the pointer's use, or use `drm_file_get_master()` if struct `drm_device.master_mutex` is not currently held and there is no other need to hold it. This prevents **master** from being freed during use.

See also **authentication** and **is_master** and the [section on primary nodes and authentication](#).

master_lookup_lock

Serializes **master**.

pid

Process that is using this file.

Must only be dereferenced under a `rcu_read_lock` or equivalent.

Updates are guarded with `dev->filelist_mutex` and reference must be dropped after a RCU grace period to accommodate lockless readers.

client_id

A unique id for `fdinfo`

magic

Authentication magic, see **authenticated**.

lhead

List of all open files of a DRM device, linked into `drm_device.filelist`. Protected by `drm_device.filelist_mutex`.

minor

`struct drm_minor` for this file.

object_idr

Mapping of mm object handles to object pointers. Used by the GEM subsystem. Protected by **table_lock**.

table_lock

Protects **object_idr**.

syncobj_idr

Mapping of sync object handles to object pointers.

syncobj_table_lock

Protects **syncobj_idr**.

filp

Pointer to the core file structure.

driver_priv

Optional pointer for driver private data. Can be allocated in `drm_driver.open` and should be freed in `drm_driver.postclose`.

fbs

List of `struct drm_framebuffer` associated with this file, using the `drm_framebuffer.filp_head` entry.

Protected by **fbs_lock**. Note that the **fbs** list holds a reference on the framebuffer object to prevent it from untimely disappearing.

fbs_lock

Protects **fbs**.

blobs

User-created blob properties; this retains a reference on the property.

Protected by **drm_mode_config.blob_lock**;

event_wait

Waitqueue for new events added to **event_list**.

pending_event_list

List of pending `struct drm_pending_event`, used to clean up pending events in case this file gets closed before the event is signalled. Uses the `drm_pending_event.pending_link` entry.

Protect by *drm_device.event_lock*.

event_list

List of *struct drm_pending_event*, ready for delivery to userspace through *drm_read()*.
Uses the *drm_pending_event.link* entry.

Protect by *drm_device.event_lock*.

event_space

Available event space to prevent userspace from exhausting kernel memory. Currently limited to the fairly arbitrary value of 4KB.

event_read_lock

Serializes *drm_read()*.

prime

Per-file buffer caches used by the PRIME buffer sharing code.

Description

This structure tracks DRM state per open file descriptor.

bool **drm_is_primary_client**(const struct *drm_file* *file_priv)
is this an open file of the primary node

Parameters

const struct *drm_file* *file_priv
DRM file

Description

Returns true if this is an open file of the primary node, i.e. *drm_file.minor* of **file_priv** is a primary minor.

See also the *section on primary nodes and authentication*.

bool **drm_is_render_client**(const struct *drm_file* *file_priv)
is this an open file of the render node

Parameters

const struct *drm_file* *file_priv
DRM file

Description

Returns true if this is an open file of the render node, i.e. *drm_file.minor* of **file_priv** is a render minor.

See also the *section on render nodes*.

bool **drm_is_accel_client**(const struct *drm_file* *file_priv)
is this an open file of the compute acceleration node

Parameters

const struct *drm_file* *file_priv
DRM file

Description

Returns true if this is an open file of the compute acceleration node, i.e. *drm_file.minor* of **file_priv** is a accel minor.

See also Introduction to compute accelerators subsystem.

struct **drm_memory_stats**

GEM object stats associated

Definition:

```
struct drm_memory_stats {  
    u64 shared;  
    u64 private;  
    u64 resident;  
    u64 purgeable;  
    u64 active;  
};
```

Members

shared

Total size of GEM objects shared between processes

private

Total size of GEM objects

resident

Total size of GEM objects backing pages

purgeable

Total size of GEM objects that can be purged (resident and not active)

active

Total size of GEM objects active on one or more engines

Description

Used by *drm_print_memory_stats()*

int **drm_open**(struct *inode* *inode, struct file *filp)

open method for DRM file

Parameters

struct **inode** *inode

device inode

struct **file** *filp

file pointer.

Description

This function must be used by drivers as their *file_operations.open* method. It looks up the correct DRM device and instantiates all the per-file resources for it. It also calls the *drm_driver.open* driver callback.

0 on success or negative errno value on failure.

Return

int **drm_release**(struct *inode* *inode, struct file *filp)
release method for DRM file

Parameters

struct inode *inode
device inode

struct file *filp
file pointer.

Description

This function must be used by drivers as their `file_operations.release` method. It frees any resources associated with the open file, and calls the *drm_driver.postclose* driver callback. If this is the last open file for the DRM device also proceeds to call the *drm_driver.lastclose* driver callback.

Always succeeds and returns 0.

Return

int **drm_release_noglobal**(struct *inode* *inode, struct file *filp)
release method for DRM file

Parameters

struct inode *inode
device inode

struct file *filp
file pointer.

Description

This function may be used by drivers as their `file_operations.release` method. It frees any resources associated with the open file prior to taking the `drm_global_mutex`, which then calls the *drm_driver.postclose* driver callback. If this is the last open file for the DRM device also proceeds to call the *drm_driver.lastclose* driver callback.

Always succeeds and returns 0.

Return

ssize_t **drm_read**(struct file *filp, char __user *buffer, size_t count, loff_t *offset)
read method for DRM file

Parameters

struct file *filp
file pointer

char __user *buffer
userspace destination pointer for the read

size_t count
count in bytes to read

loff_t *offset
offset to read

Description

This function must be used by drivers as their `file_operations.read` method if they use DRM events for asynchronous signalling to userspace. Since events are used by the KMS API for vblank and page flip completion this means all modern display drivers must use it.

offset is ignored, DRM events are read like a pipe. Polling support is provided by [`drm_poll\(\)`](#).

This function will only ever read a full event. Therefore userspace must supply a big enough buffer to fit any event to ensure forward progress. Since the maximum event space is currently 4K it's recommended to just use that for safety.

Number of bytes read (always aligned to full events, and can be 0) or a negative error code on failure.

Return

`__poll_t` **drm_poll**(struct file *filp, struct poll_table_struct *wait)
poll method for DRM file

Parameters

struct file *filp
file pointer

struct poll_table_struct *wait
poll waiter table

Description

This function must be used by drivers as their `file_operations.read` method if they use DRM events for asynchronous signalling to userspace. Since events are used by the KMS API for vblank and page flip completion this means all modern display drivers must use it.

See also [`drm_read\(\)`](#).

Mask of POLL flags indicating the current status of the file.

Return

int **drm_event_reserve_init_locked**(struct [`drm_device`](#) *dev, struct [`drm_file`](#) *file_priv, struct [`drm_pending_event`](#) *p, struct [`drm_event`](#) *e)
init a DRM event and reserve space for it

Parameters

struct drm_device *dev
DRM device

struct drm_file *file_priv
DRM file private data

struct drm_pending_event *p
tracking structure for the pending event

struct drm_event *e
actual event data to deliver to userspace

Description

This function prepares the passed in event for eventual delivery. If the event doesn't get delivered (because the IOCTL fails later on, before queuing up anything) then the event must be

cancelled and freed using `drm_event_cancel_free()`. Successfully initialized events should be sent out using `drm_send_event()` or `drm_send_event_locked()` to signal completion of the asynchronous event to userspace.

If callers embedded **p** into a larger structure it must be allocated with `kmalloc` and **p** must be the first member element.

This is the locked version of `drm_event_reserve_init()` for callers which already hold `drm_device.event_lock`.

0 on success or a negative error code on failure.

Return

int **drm_event_reserve_init**(struct *drm_device* *dev, struct *drm_file* *file_priv, struct *drm_pending_event* *p, struct *drm_event* *e)

init a DRM event and reserve space for it

Parameters

struct drm_device *dev

DRM device

struct drm_file *file_priv

DRM file private data

struct drm_pending_event *p

tracking structure for the pending event

struct drm_event *e

actual event data to deliver to userspace

Description

This function prepares the passed in event for eventual delivery. If the event doesn't get delivered (because the IOCTL fails later on, before queuing up anything) then the even must be cancelled and freed using `drm_event_cancel_free()`. Successfully initialized events should be sent out using `drm_send_event()` or `drm_send_event_locked()` to signal completion of the asynchronous event to userspace.

If callers embedded **p** into a larger structure it must be allocated with `kmalloc` and **p** must be the first member element.

Callers which already hold `drm_device.event_lock` should use `drm_event_reserve_init_locked()` instead.

0 on success or a negative error code on failure.

Return

void **drm_event_cancel_free**(struct *drm_device* *dev, struct *drm_pending_event* *p)

free a DRM event and release its space

Parameters

struct drm_device *dev

DRM device

struct drm_pending_event *p

tracking structure for the pending event

Description

This function frees the event **p** initialized with `drm_event_reserve_init()` and releases any allocated space. It is used to cancel an event when the nonblocking operation could not be submitted and needed to be aborted.

```
void drm_send_event_timestamp_locked(struct drm_device *dev, struct drm_pending_event
                                     *e, ktime_t timestamp)
```

send DRM event to file descriptor

Parameters

struct *drm_device* *dev
DRM device

struct *drm_pending_event* *e
DRM event to deliver

ktime_t timestamp
timestamp to set for the fence event in kernel's CLOCK_MONOTONIC time domain

Description

This function sends the event **e**, initialized with `drm_event_reserve_init()`, to its associated userspace DRM file. Callers must already hold `drm_device.event_lock`.

Note that the core will take care of unlinking and disarming events when the corresponding DRM file is closed. Drivers need not worry about whether the DRM file for this event still exists and can call this function upon completion of the asynchronous work unconditionally.

```
void drm_send_event_locked(struct drm_device *dev, struct drm_pending_event *e)
    send DRM event to file descriptor
```

Parameters

struct *drm_device* *dev
DRM device

struct *drm_pending_event* *e
DRM event to deliver

Description

This function sends the event **e**, initialized with `drm_event_reserve_init()`, to its associated userspace DRM file. Callers must already hold `drm_device.event_lock`, see `drm_send_event()` for the unlocked version.

Note that the core will take care of unlinking and disarming events when the corresponding DRM file is closed. Drivers need not worry about whether the DRM file for this event still exists and can call this function upon completion of the asynchronous work unconditionally.

```
void drm_send_event(struct drm_device *dev, struct drm_pending_event *e)
    send DRM event to file descriptor
```

Parameters

struct *drm_device* *dev
DRM device

struct *drm_pending_event* *e
DRM event to deliver

Description

This function sends the event **e**, initialized with [drm_event_reserve_init\(\)](#), to its associated userspace DRM file. This function acquires [drm_device.event_lock](#), see [drm_send_event_locked\(\)](#) for callers which already hold this lock.

Note that the core will take care of unlinking and disarming events when the corresponding DRM file is closed. Drivers need not worry about whether the DRM file for this event still exists and can call this function upon completion of the asynchronous work unconditionally.

```
void drm_print_memory_stats(struct drm\_printer *p, const struct drm\_memory\_stats *stats,  
                           enum drm\_gem\_object\_status supported_status, const char  
                           *region)
```

A helper to print memory stats

Parameters

struct [drm_printer](#) *p

The printer to print output to

const struct [drm_memory_stats](#) *stats

The collected memory stats

enum [drm_gem_object_status](#) supported_status

Bitmask of optional stats which are available

const char *region

The memory region

```
void drm_show_memory_stats(struct drm\_printer *p, struct drm\_file *file)
```

Helper to collect and show standard fdinfo memory stats

Parameters

struct [drm_printer](#) *p

the printer to print output to

struct [drm_file](#) *file

the DRM file

Description

Helper to iterate over GEM objects with a handle allocated in the specified file.

```
void drm_show_fdinfo(struct seq_file *m, struct file *f)
```

helper for drm file fops

Parameters

struct seq_file *m

output stream

struct file *f

the device file instance

Description

Helper to implement fdinfo, for userspace to query usage stats, etc, of a process using the GPU. See also [drm_driver.show_fdinfo](#).

For text output format description please see [DRM client usage stats](#)

2.3 Misc Utilities

2.3.1 Printer

A simple wrapper for `dev_printk()`, `seq_printf()`, etc. Allows same debug code to be used for both debugfs and printk logging.

For example:

```
void log_some_info(struct drm_printer *p)
{
    drm_printf(p, "foo=%d\n", foo);
    drm_printf(p, "bar=%d\n", bar);
}

#ifdef CONFIG_DEBUG_FS
void debugfs_show(struct seq_file *f)
{
    struct drm_printer p = drm_seq_file_printer(f);
    log_some_info(&p);
}
#endif

void some_other_function(...)
{
    struct drm_printer p = drm_info_printer(drm->dev);
    log_some_info(&p);
}
```

struct **drm_printer**
 drm output “stream”

Definition:

```
struct drm_printer {
};
```

Members

Description

Do not use struct members directly. Use `drm_printer_seq_file()`, `drm_printer_info()`, etc to initialize. And *drm_printf()* for output.

void **drm_vprintf**(struct *drm_printer* *p, const char *fmt, va_list *va)
 print to a *drm_printer* stream

Parameters

struct **drm_printer** *p
 the *drm_printer*

const char *fmt
 format string

va_list *va

the va_list

drm_printf_indent

drm_printf_indent (printer, indent, fmt, ...)

Print to a *drm_printer* stream with indentation

Parameters

printer

DRM printer

indent

Tab indentation level (max 5)

fmt

Format string

...

variable arguments

struct **drm_print_iterator**

local struct used with *drm_printer_coredump*

Definition:

```
struct drm_print_iterator {
    void *data;
    ssize_t start;
    ssize_t remain;
};
```

Members

data

Pointer to the devcoredump output buffer

start

The offset within the buffer to start writing

remain

The number of bytes to write for this iteration

struct *drm_printer* **drm_coredump_printer**(struct *drm_print_iterator* *iter)

construct a *drm_printer* that can output to a buffer from the read function for devcoredump

Parameters

struct drm_print_iterator *iter

A pointer to a *struct drm_print_iterator* for the read instance

Description

This wrapper extends *drm_printf()* to work with a *dev_coredumpm()* callback function. The passed in *drm_print_iterator* struct contains the buffer pointer, size and offset as passed in from *devcoredump*.

For example:

```

void coredump_read(char *buffer, loff_t offset, size_t count,
                  void *data, size_t datalen)
{
    struct drm_print_iterator iter;
    struct drm_printer p;

    iter.data = buffer;
    iter.start = offset;
    iter.remain = count;

    p = drm_coredump_printer(&iter);

    drm_printf(p, "foo=%d\n", foo);
}

void makecoredump(...)
{
    ...
    dev_coredumpm(dev, THIS_MODULE, data, 0, GFP_KERNEL,
                  coredump_read, ...)
}

```

Return

The *drm_printer* object

struct *drm_printer* **drm_seq_file_printer**(struct seq_file *f)
 construct a *drm_printer* that outputs to seq_file

Parameters

struct seq_file *f
 the struct seq_file to output to

Return

The *drm_printer* object

struct *drm_printer* **drm_info_printer**(struct device *dev)
 construct a *drm_printer* that outputs to dev_printk()

Parameters

struct device *dev
 the struct device pointer

Return

The *drm_printer* object

struct *drm_printer* **drm_debug_printer**(const char *prefix)
 construct a *drm_printer* that outputs to pr_debug()

Parameters

const char *prefix
 debug output prefix

Return

The *drm_printer* object

struct *drm_printer* **drm_err_printer**(const char *prefix)
construct a *drm_printer* that outputs to pr_err()

Parameters

const char *prefix
debug output prefix

Return

The *drm_printer* object

enum **drm_debug_category**
The DRM debug categories

Constants

DRM_UT_CORE
Used in the generic drm code: *drm_ioctl.c*, *drm_mm.c*, *drm_memory.c*, ...

DRM_UT_DRIVER
Used in the vendor specific part of the driver: *i915*, *radeon*, ... macro.

DRM_UT_KMS
Used in the modesetting code.

DRM_UT_PRIME
Used in the prime code.

DRM_UT_ATOMIC
Used in the atomic code.

DRM_UT_VBL
Used for verbose debug message in the vblank code.

DRM_UT_STATE
Used for verbose atomic state debugging.

DRM_UT_LEASE
Used in the lease code.

DRM_UT_DP
Used in the DP code.

DRM_UT_DRMRES
Used in the drm managed resources code.

Description

Each of the DRM debug logging macros use a specific category, and the logging is filtered by the *drm.debug* module parameter. This enum specifies the values for the interface.

Each *DRM_DEBUG_<CATEGORY>* macro logs to *DRM_UT_<CATEGORY>* category, except *DRM_DEBUG()* logs to *DRM_UT_CORE*.

Enabling verbose debug messages is done through the *drm.debug* parameter, each category being enabled by a bit:

- `drm.debug=0x1` will enable CORE messages
- `drm.debug=0x2` will enable DRIVER messages
- `drm.debug=0x3` will enable CORE and DRIVER messages
- ...
- `drm.debug=0x1ff` will enable all messages

An interesting feature is that it's possible to enable verbose logging at run-time by echoing the debug value in its sysfs node:

```
# echo 0xf > /sys/module/drm/parameters/debug
```

DRM_DEV_ERROR

`DRM_DEV_ERROR (dev, fmt, ...)`

Error output.

Parameters

dev

device pointer

fmt

printf() like format string.

...

variable arguments

NOTE

this is deprecated in favor of `drm_err()` or `dev_err()`.

DRM_DEV_ERROR_RATELIMITED

`DRM_DEV_ERROR_RATELIMITED (dev, fmt, ...)`

Rate limited error output.

Parameters

dev

device pointer

fmt

printf() like format string.

...

variable arguments

NOTE

this is deprecated in favor of `drm_err_ratelimited()` or `dev_err_ratelimited()`.

Description

Like `DRM_ERROR()` but won't flood the log.

DRM_DEV_DEBUG

`DRM_DEV_DEBUG (dev, fmt, ...)`

Debug output for generic drm code

Parameters

dev

device pointer

fmt

printf() like format string.

...

variable arguments

NOTE

this is deprecated in favor of `drm_dbg_core()`.

DRM_DEV_DEBUG_DRIVER

`DRM_DEV_DEBUG_DRIVER (dev, fmt, ...)`

Debug output for vendor specific part of the driver

Parameters

dev

device pointer

fmt

printf() like format string.

...

variable arguments

NOTE

this is deprecated in favor of `drm_dbg()` or `dev_dbg()`.

DRM_DEV_DEBUG_KMS

`DRM_DEV_DEBUG_KMS (dev, fmt, ...)`

Debug output for modesetting code

Parameters

dev

device pointer

fmt

printf() like format string.

...

variable arguments

NOTE

this is deprecated in favor of `drm_dbg_kms()`.

void **drm_puts**(struct *drm_printer* *p, const char *str)
print a const string to a *drm_printer* stream

Parameters

struct drm_printer *p
the drm printer

const char *str
const string

Description

Allow *drm_printer* types that have a constant string option to use it.

void **drm_printf**(struct *drm_printer* *p, const char *f, ...)
print to a *drm_printer* stream

Parameters

struct drm_printer *p
the *drm_printer*

const char *f
format string

...
variable arguments

void **drm_print_bits**(struct *drm_printer* *p, unsigned long value, const char *const bits[],
unsigned int nbits)
print bits to a *drm_printer* stream

Parameters

struct drm_printer *p
the *drm_printer*

unsigned long value
field value.

const char * const bits[]
Array with bit names.

unsigned int nbits
Size of bit names array.

Description

Print bits (in flag fields for example) in human readable form.

void **drm_print_regset32**(struct *drm_printer* *p, struct debugfs_regset32 *regset)
print the contents of registers to a *drm_printer* stream.

Parameters

struct drm_printer *p
the drm printer

struct debugfs_regset32 *regset
the list of registers to print.

Description

Often in driver debug, it's useful to be able to either capture the contents of registers in the steady state using debugfs or at specific points during operation. This lets the driver have a single list of registers for both.

2.3.2 Utilities

Macros and inline functions that does not naturally belong in other places

for_each_if

`for_each_if (condition)`

helper for handling conditionals in various `for_each` macros

Parameters

condition

The condition to check

Description

Typical use:

```
#define for_each_foo_bar(x, y) \
    list_for_each_entry(x, y->list, head) \
        for_each_if(x->something == SOMETHING)
```

The `for_each_if()` macro makes the use of `for_each_foo_bar()` less error prone.

bool **drm_can_sleep**(void)

returns true if currently okay to sleep

Parameters

void

no arguments

Description

This function shall not be used in new code. The check for running in atomic context may not work - see `linux/preempt.h`.

FIXME: All users of `drm_can_sleep` should be removed (see [TODO list](#))

Return

False if kgdb is active, we are in atomic context or irqs are disabled.

2.4 Unit testing

2.4.1 KUnit

KUnit (Kernel unit testing framework) provides a common framework for unit tests within the Linux kernel.

This section covers the specifics for the DRM subsystem. For general information about KUnit, please refer to [Documentation/dev-tools/kunit/start.rst](#).

How to run the tests?

In order to facilitate running the test suite, a configuration file is present in `drivers/gpu/drm/tests/.kunitconfig`. It can be used by `kunit.py` as follows:

```
$ ./tools/testing/kunit/kunit.py run --kunitconfig=drivers/gpu/drm/tests \
    --kconfig_add CONFIG_VIRTIO_UML=y \
    --kconfig_add CONFIG_UML_PCI_OVER_VIRTIO=y
```

Note: The configuration included in `.kunitconfig` should be as generic as possible. `CONFIG_VIRTIO_UML` and `CONFIG_UML_PCI_OVER_VIRTIO` are not included in it because they are only required for User Mode Linux.

2.5 Legacy Support Code

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

2.5.1 Legacy Suspend/Resume

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

`int (*suspend) (struct drm_device *, pm_message_t state); int (*resume) (struct drm_device *);` Those are legacy suspend and resume methods which *only* work with the legacy shadow-attach driver registration functions. New driver should use the power management interface provided by their bus type (usually through the `struct device_driver dev_pm_ops`) and set these methods to `NULL`.

2.5.2 Legacy DMA Services

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

DRM 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 Manager (TTM) and Graphics Execution Manager (GEM). TTM was the first DRM memory manager to be developed and tried to be a one-size-fits-them all solution. It provides a single userspace API to accommodate the need of all hardware, supporting both Unified Memory Architecture (UMA) devices and devices with dedicated video RAM (i.e. most discrete video cards). This resulted in a large, complex piece of code that turned out to be hard to use for driver development.

GEM started as an Intel-sponsored project in reaction to TTM's complexity. Its design philosophy is completely different: instead of providing a solution to every graphics memory-related problems, GEM identified common code between drivers and created a support library to share it. GEM has simpler initialization and execution requirements than TTM, but has no video RAM management capabilities and is thus limited to UMA devices.

3.1 The Translation Table Manager (TTM)

TTM is a memory manager for accelerator devices with dedicated memory.

The basic idea is that resources are grouped together in buffer objects of certain size and TTM handles lifetime, movement and CPU mappings of those objects.

TODO: Add more design background and information here.

enum **ttm_caching**

CPU caching and BUS snooping behavior.

Constants

ttm_uncached

Most defensive option for device mappings, don't even allow write combining.

ttm_write_combined

Don't cache read accesses, but allow at least writes to be combined.

ttm_cached

Fully cached like normal system memory, requires that devices snoop the CPU cache on accesses.

3.1.1 TTM device object reference

struct **ttm_global**

Buffer object driver global data.

Definition:

```
struct ttm_global {
    struct page *dummy_read_page;
    struct list_head device_list;
    atomic_t bo_count;
};
```

Members

dummy_read_page

Pointer to a dummy page used for mapping requests of unpopulated pages. Constant after init.

device_list

List of buffer object devices. Protected by `ttm_global_mutex`.

bo_count

Number of buffer objects allocated by devices.

struct **ttm_device**

Buffer object driver device-specific data.

Definition:

```
struct ttm_device {
    struct list_head device_list;
    const struct ttm_device_funcs *funcs;
    struct ttm_resource_manager sysman;
    struct ttm_resource_manager *man_drv[TTM_NUM_MEM_TYPES];
    struct drm_vma_offset_manager *vma_manager;
    struct ttm_pool pool;
    spinlock_t lru_lock;
    struct list_head pinned;
    struct address_space *dev_mapping;
    struct workqueue_struct *wq;
};
```

Members

device_list

Our entry in the global device list. Constant after bo device init

funcs

Function table for the device. Constant after bo device init

sysman

Resource manager for the system domain. Access via `ttm_manager_type`.

man_drv

An array of `resource_managers`, one per resource type.

vma_manager

Address space manager for finding BOs to mmap.

pool

page pool for the device.

lru_lock

Protection for the per manager LRU and ddestroy lists.

pinned

Buffer objects which are pinned and so not on any LRU list.

dev_mapping

A pointer to the struct address_space for invalidating CPU mappings on buffer move. Protected by load/unload sync.

wq

Work queue structure for the delayed delete workqueue.

```
int ttm_device_init(struct ttm_device *bdev, const struct ttm_device_funcs *funcs, struct
                    device *dev, struct address_space *mapping, struct
                    drm_vma_offset_manager *vma_manager, bool use_dma_alloc, bool
                    use_dma32)
```

Parameters**struct ttm_device *bdev**

A pointer to a *struct ttm_device* to initialize.

const struct ttm_device_funcs *funcs

Function table for the device.

struct device *dev

The core kernel device pointer for DMA mappings and allocations.

struct address_space *mapping

The address space to use for this bo.

struct drm_vma_offset_manager *vma_manager

A pointer to a vma manager.

bool use_dma_alloc

If coherent DMA allocation API should be used.

bool use_dma32

If we should use GFP_DMA32 for device memory allocations.

Description

Initializes a *struct ttm_device*:

Return

!0: Failure.

3.1.2 TTM resource placement reference

struct **ttm_place**

Definition:

```
struct ttm_place {
    unsigned fpfn;
    unsigned lpfn;
    uint32_t mem_type;
    uint32_t flags;
};
```

Members

fpfn

first valid page frame number to put the object

lpfn

last valid page frame number to put the object

mem_type

One of TTM_PL_* where the resource should be allocated from.

flags

memory domain and caching flags for the object

Description

Structure indicating a possible place to put an object.

struct **ttm_placement**

Definition:

```
struct ttm_placement {
    unsigned num_placement;
    const struct ttm_place *placement;
    unsigned num_busy_placement;
    const struct ttm_place *busy_placement;
};
```

Members

num_placement

number of preferred placements

placement

preferred placements

num_busy_placement

number of preferred placements when need to evict buffer

busy_placement

preferred placements when need to evict buffer

Description

Structure indicating the placement you request for an object.

3.1.3 TTM resource object reference

struct **ttm_resource_manager**

Definition:

```
struct ttm_resource_manager {
    bool use_type;
    bool use_tt;
    struct ttm_device *bdev;
    uint64_t size;
    const struct ttm_resource_manager_func *func;
    spinlock_t move_lock;
    struct dma_fence *move;
    struct list_head lru[TTM_MAX_BO_PRIORITY];
    uint64_t usage;
};
```

Members

use_type

The memory type is enabled.

use_tt

If a TT object should be used for the backing store.

bdev

ttm device this manager belongs to

size

Size of the managed region.

func

structure pointer implementing the range manager. See above

move_lock

lock for move fence

move

The fence of the last pipelined move operation.

lru

The lru list for this memory type.

usage

How much of the resources are used, protected by the bdev->lru_lock.

Description

This structure is used to identify and manage memory types for a device.

struct **ttm_bus_placement**

Definition:

```
struct ttm_bus_placement {
    void *addr;
    phys_addr_t offset;
```

```
bool is_iomem;
enum ttm_caching      caching;
};
```

Members

addr

mapped virtual address

offset

physical addr

is_iomem

is this io memory ?

caching

See [enum ttm_caching](#)

Description

Structure indicating the bus placement of an object.

struct **ttm_resource**

Definition:

```
struct ttm_resource {
    unsigned long start;
    size_t size;
    uint32_t mem_type;
    uint32_t placement;
    struct ttm_bus_placement bus;
    struct ttm_buffer_object *bo;
    struct list_head lru;
};
```

Members

start

Start of the allocation.

size

Actual size of resource in bytes.

mem_type

Resource type of the allocation.

placement

Placement flags.

bus

Placement on io bus accessible to the CPU

bo

weak reference to the BO, protected by `ttm_device::lru_lock`

lru

Least recently used list, see [ttm_resource_manager.lru](#)

Description

Structure indicating the placement and space resources used by a buffer object.

struct **ttm_resource_cursor**

Definition:

```
struct ttm_resource_cursor {  
    unsigned int priority;  
};
```

Members

priority

the current priority

Description

Cursor to iterate over the resources in a manager.

struct **ttm_lru_bulk_move_pos**

Definition:

```
struct ttm_lru_bulk_move_pos {  
    struct ttm_resource *first;  
    struct ttm_resource *last;  
};
```

Members

first

first res in the bulk move range

last

last res in the bulk move range

Description

Range of resources for a lru bulk move.

struct **ttm_lru_bulk_move**

Definition:

```
struct ttm_lru_bulk_move {  
    struct ttm_lru_bulk_move_pos pos[TTM_NUM_MEM_TYPES][TTM_MAX_BO_PRIORITY];  
};
```

Members

pos

first/last lru entry for resources in the each domain/priority

Description

Container for the current bulk move state. Should be used with [*ttm_lru_bulk_move_init\(\)*](#) and [*ttm_bo_set_bulk_move\(\)*](#).

struct **ttm_kmap_iter_iomap**

Specialization for a struct `io_mapping` + struct `sg_table` backed *struct ttm_resource*.

Definition:

```
struct ttm_kmap_iter_iomap {
    struct ttm_kmap_iter base;
    struct io_mapping *iomap;
    struct sg_table *st;
    resource_size_t start;
    struct {
        struct scatterlist *sg;
        pgoff_t i;
        pgoff_t end;
        pgoff_t offs;
    } cache;
};
```

Members

base

Embedded struct `ttm_kmap_iter` providing the usage interface.

iomap

struct `io_mapping` representing the underlying linear `io_memory`.

st

`sg_table` into **iomap**, representing the memory of the *struct ttm_resource*.

start

Offset that needs to be subtracted from **st** to make `sg_dma_address(st->sgl) - start == 0` for **iomap** start.

cache

Scatterlist traversal cache for fast lookups.

cache.sg

Pointer to the currently cached scatterlist segment.

cache.i

First index of **sg**. `PAGE_SIZE` granularity.

cache.end

Last index + 1 of **sg**. `PAGE_SIZE` granularity.

cache.offs

First offset into **iomap** of **sg**. `PAGE_SIZE` granularity.

struct **ttm_kmap_iter_linear_io**

Iterator specialization for linear `io`

Definition:

```
struct ttm_kmap_iter_linear_io {
    struct ttm_kmap_iter base;
    struct iosys_map dmap;
};
```

```
bool needs_unmap;
};
```

Members

base

The base iterator

dmap

Points to the starting address of the region

needs_unmap

Whether we need to unmap on fini

void **ttm_resource_manager_set_used**(struct *ttm_resource_manager* *man, bool used)

Parameters

struct ttm_resource_manager *man

A memory manager object.

bool used

usage state to set.

Description

Set the manager in use flag. If disabled the manager is no longer used for object placement.

bool **ttm_resource_manager_used**(struct *ttm_resource_manager* *man)

Parameters

struct ttm_resource_manager *man

Manager to get used state for

Description

Get the in use flag for a manager.

Return

true is used, false if not.

void **ttm_resource_manager_cleanup**(struct *ttm_resource_manager* *man)

Parameters

struct ttm_resource_manager *man

A memory manager object.

Description

Cleanup the move fences from the memory manager object.

ttm_resource_manager_for_each_res

ttm_resource_manager_for_each_res (man, cursor, res)

iterate over all resources

Parameters

man

the resource manager

cursor

struct ttm_resource_cursor for the current position

res

the current resource

Description

Iterate over all the evictable resources in a resource manager.

void **ttm_lru_bulk_move_init**(struct *ttm_lru_bulk_move* *bulk)

initialize a bulk move structure

Parameters

struct ttm_lru_bulk_move *bulk

the structure to init

Description

For now just memset the structure to zero.

void **ttm_lru_bulk_move_tail**(struct *ttm_lru_bulk_move* *bulk)

bulk move range of resources to the LRU tail.

Parameters

struct ttm_lru_bulk_move *bulk

bulk move structure

Description

Bulk move BOs to the LRU tail, only valid to use when driver makes sure that resource order never changes. Should be called with *ttm_device.lru_lock* held.

void **ttm_resource_init**(struct ttm_buffer_object *bo, const struct *ttm_place* *place, struct *ttm_resource* *res)

resource object constructure

Parameters

struct ttm_buffer_object *bo

buffer object this resources is allocated for

const struct ttm_place *place

placement of the resource

struct ttm_resource *res

the resource object to inistilize

Description

Initialize a new resource object. Counterpart of *ttm_resource_fini()*.

void **ttm_resource_fini**(struct *ttm_resource_manager* *man, struct *ttm_resource* *res)

resource destructor

Parameters

struct ttm_resource_manager *man
the resource manager this resource belongs to

struct ttm_resource *res
the resource to clean up

Description

Should be used by resource manager backends to clean up the TTM resource objects before freeing the underlying structure. Makes sure the resource is removed from the LRU before destruction. Counterpart of *ttm_resource_init()*.

void **ttm_resource_manager_init**(struct *ttm_resource_manager* *man, struct *ttm_device* *bdev, uint64_t size)

Parameters

struct ttm_resource_manager *man
memory manager object to init

struct ttm_device *bdev
ttm device this manager belongs to

uint64_t size
size of managed resources in arbitrary units

Description

Initialise core parts of a manager object.

uint64_t **ttm_resource_manager_usage**(struct *ttm_resource_manager* *man)

Parameters

struct ttm_resource_manager *man
A memory manager object.

Description

Return how many resources are currently used.

void **ttm_resource_manager_debug**(struct *ttm_resource_manager* *man, struct *drm_printer* *p)

Parameters

struct ttm_resource_manager *man
manager type to dump.

struct drm_printer *p
printer to use for debug.

struct ttm_kmap_iter ***ttm_kmap_iter_iomap_init**(struct *ttm_kmap_iter_iomap* *iter_io, struct io_mapping *iomap, struct sg_table *st, resource_size_t start)

Initialize a *struct ttm_kmap_iter_iomap*

Parameters

struct ttm_kmap_iter_iomap *iter_io
The *struct ttm_kmap_iter_iomap* to initialize.

struct io_mapping *iomap

The struct io_mapping representing the underlying linear io_memory.

struct sg_table *st

sg_table into **iomap**, representing the memory of the *struct ttm_resource*.

resource_size_t start

Offset that needs to be subtracted from **st** to make `sg_dma_address(st->sgl) - start == 0` for **iomap** start.

Return

Pointer to the embedded struct ttm_kmap_iter.

void **ttm_resource_manager_create_debugfs**(struct *ttm_resource_manager* *man, struct dentry *parent, const char *name)

Create debugfs entry for specified resource manager.

Parameters

struct ttm_resource_manager *man

The TTM resource manager for which the debugfs stats file be creates

struct dentry * parent

debugfs directory in which the file will reside

const char *name

The filename to create.

Description

This function setups up a debugfs file that can be used to look at debug statistics of the specified ttm_resource_manager.

3.1.4 TTM TT object reference

struct ttm_tt

This is a structure holding the pages, caching- and aperture binding status for a buffer object that isn't backed by fixed (VRAM / AGP) memory.

Definition:

```
struct ttm_tt {
    struct page **pages;
#define TTM TT_FLAG_SWAPPED          BIT(0);
#define TTM TT_FLAG_ZERO_ALLOC       BIT(1);
#define TTM TT_FLAG_EXTERNAL         BIT(2);
#define TTM TT_FLAG_EXTERNAL_MAPPABLE BIT(3);
#define TTM TT_FLAG_DECRYPTED         BIT(4);
#define TTM TT_FLAG_PRIV_POPULATED   BIT(5);
    uint32_t page_flags;
    uint32_t num_pages;
    struct sg_table *sg;
    dma_addr_t *dma_address;
    struct file *swap_storage;
```

```
enum ttm_caching caching;
};
```

Members

pages

Array of pages backing the data.

page_flags

The page flags.

Supported values:

TTM_TT_FLAG_SWAPPED: Set by TTM when the pages have been unpopulated and swapped out by TTM. Calling *ttm_tt_populate()* will then swap the pages back in, and unset the flag. Drivers should in general never need to touch this.

TTM_TT_FLAG_ZERO_ALLOC: Set if the pages will be zeroed on allocation.

TTM_TT_FLAG_EXTERNAL: Set if the underlying pages were allocated externally, like with dma-buf or userptr. This effectively disables TTM swapping out such pages. Also important is to prevent TTM from ever directly mapping these pages.

Note that enum *ttm_bo_type.ttm_bo_type_sg* objects will always enable this flag.

TTM_TT_FLAG_EXTERNAL_MAPPABLE: Same behaviour as TTM_TT_FLAG_EXTERNAL, but with the reduced restriction that it is still valid to use TTM to map the pages directly. This is useful when implementing a *ttm_tt* backend which still allocates driver owned pages underneath (say with shmem).

Note that since this also implies TTM_TT_FLAG_EXTERNAL, the usage here should always be:

```
page_flags = TTM_TT_FLAG_EXTERNAL |  
TTM_TT_FLAG_EXTERNAL_MAPPABLE;
```

TTM_TT_FLAG_DECRYPTED: The mapped ttm pages should be marked as not encrypted. The framework will try to match what the dma layer is doing, but note that it is a little fragile because ttm page fault handling abuses the DMA api a bit and *dma_map_attrs* can't be used to assure pgprot always matches.

TTM_TT_FLAG_PRIV_POPULATED: TTM internal only. DO NOT USE. This is set by TTM after *ttm_tt_populate()* has successfully returned, and is then unset when TTM calls *ttm_tt_unpopulate()*.

num_pages

Number of pages in the page array.

sg

for SG objects via dma-buf.

dma_address

The DMA (bus) addresses of the pages.

swap_storage

Pointer to shmem struct file for swap storage.

caching

The current caching state of the pages, see *enum ttm_caching*.

struct **ttm_kmap_iter_tt**

Specialization of a mappig iterator for a tt.

Definition:

```
struct ttm_kmap_iter_tt {
    struct ttm_kmap_iter base;
    struct ttm_tt *tt;
    pgprot_t prot;
};
```

Members

base

Embedded struct ttm_kmap_iter providing the usage interface

tt

Cached *struct ttm_tt*.

prot

Cached page protection for mapping.

int **ttm_tt_create**(struct ttm_buffer_object *bo, bool zero_alloc)

Parameters

struct ttm_buffer_object *bo

pointer to a struct ttm_buffer_object

bool zero_alloc

true if allocated pages needs to be zeroed

Description

Make sure we have a TTM structure allocated for the given BO. No pages are actually allocated.

int **ttm_tt_init**(struct *ttm_tt* *ttm, struct ttm_buffer_object *bo, uint32_t page_flags, enum *ttm_caching* caching, unsigned long extra_pages)

Parameters

struct ttm_tt *ttm

The *struct ttm_tt*.

struct ttm_buffer_object *bo

The buffer object we create the ttm for.

uint32_t page_flags

Page flags as identified by TTM_TT_FLAG_XX flags.

enum ttm_caching caching

the desired caching state of the pages

unsigned long extra_pages

Extra pages needed for the driver.

Description

Create a *struct ttm_tt* to back data with system memory pages. No pages are actually allocated.

Return

NULL: Out of memory.

void **ttm_tt_fini**(struct *ttm_tt* *ttm)

Parameters

struct ttm_tt *ttm
the ttm_tt structure.

Description

Free memory of ttm_tt structure

void **ttm_tt_destroy**(struct *ttm_device* *bdev, struct *ttm_tt* *ttm)

Parameters

struct ttm_device *bdev
the ttm_device this object belongs to

struct ttm_tt *ttm
The *struct ttm_tt*.

Description

Unbind, unpopulate and destroy common *struct ttm_tt*.

int **ttm_tt_swapin**(struct *ttm_tt* *ttm)

Parameters

struct ttm_tt *ttm
The *struct ttm_tt*.

Description

Swap in a previously swap out ttm_tt.

int **ttm_tt_populate**(struct *ttm_device* *bdev, struct *ttm_tt* *ttm, struct ttm_operation_ctx *ctx)

allocate pages for a ttm

Parameters

struct ttm_device *bdev
the ttm_device this object belongs to

struct ttm_tt *ttm
Pointer to the ttm_tt structure

struct ttm_operation_ctx *ctx
operation context for populating the tt object.

Description

Calls the driver method to allocate pages for a ttm

void **ttm_tt_unpopulate**(struct *ttm_device* *bdev, struct *ttm_tt* *ttm)
free pages from a ttm

Parameters

struct ttm_device *bdev

the ttm_device this object belongs to

struct ttm_tt *ttm

Pointer to the ttm_tt structure

Description

Calls the driver method to free all pages from a ttm

void **ttm_tt_mark_for_clear**(struct *ttm_tt* *ttm)

Mark pages for clearing on populate.

Parameters

struct ttm_tt *ttm

Pointer to the ttm_tt structure

Description

Marks pages for clearing so that the next time the page vector is populated, the pages will be cleared.

struct *ttm_tt* ***ttm_agp_tt_create**(struct ttm_buffer_object *bo, struct agp_bridge_data *bridge, uint32_t page_flags)

Parameters

struct ttm_buffer_object *bo

Buffer object we allocate the ttm for.

struct agp_bridge_data *bridge

The agp bridge this device is sitting on.

uint32_t page_flags

Page flags as identified by TTM_TT_FLAG_XX flags.

Description

Create a TTM backend that uses the indicated AGP bridge as an aperture for TT memory. This function uses the linux agpgart interface to bind and unbind memory backing a ttm_tt.

struct ttm_kmap_iter ***ttm_kmap_iter_tt_init**(struct *ttm_kmap_iter_tt* *iter_tt, struct *ttm_tt* *tt)

Initialize a *struct ttm_kmap_iter_tt*

Parameters

struct ttm_kmap_iter_tt *iter_tt

The *struct ttm_kmap_iter_tt* to initialize.

struct ttm_tt *tt

Struct ttm_tt holding page pointers of the *struct ttm_resource*.

Return

Pointer to the embedded struct ttm_kmap_iter.

3.1.5 TTM page pool reference

struct **ttm_pool_type**

Pool for a certain memory type

Definition:

```
struct ttm_pool_type {
    struct ttm_pool *pool;
    unsigned int order;
    enum ttm_caching caching;
    struct list_head shrinker_list;
    spinlock_t lock;
    struct list_head pages;
};
```

Members

pool

the pool we belong to, might be NULL for the global ones

order

the allocation order our pages have

caching

the caching type our pages have

shrinker_list

our place on the global shrinker list

lock

protection of the page list

pages

the list of pages in the pool

struct **ttm_pool**

Pool for all caching and orders

Definition:

```
struct ttm_pool {
    struct device *dev;
    int nid;
    bool use_dma_alloc;
    bool use_dma32;
    struct {
        struct ttm_pool_type orders[NR_PAGE_ORDERS];
    } caching[TTM_NUM_CACHING_TYPES];
};
```

Members

dev

the device we allocate pages for

nid

which numa node to use

use_dma_alloc

if coherent DMA allocations should be used

use_dma32

if GFP_DMA32 should be used

caching

pools for each caching/order

int **ttm_pool_alloc**(struct *ttm_pool* *pool, struct *ttm_tt* *tt, struct ttm_operation_ctx *ctx)

Fill a ttm_tt object

Parameters

struct ttm_pool *pool

ttm_pool to use

struct ttm_tt *tt

ttm_tt object to fill

struct ttm_operation_ctx *ctx

operation context

Description

Fill the ttm_tt object with pages and also make sure to DMA map them when necessary.

Return

0 on successe, negative error code otherwise.

void **ttm_pool_free**(struct *ttm_pool* *pool, struct *ttm_tt* *tt)

Free the backing pages from a ttm_tt object

Parameters

struct ttm_pool *pool

Pool to give pages back to.

struct ttm_tt *tt

ttm_tt object to unpopulate

Description

Give the packing pages back to a pool or free them

void **ttm_pool_init**(struct *ttm_pool* *pool, struct device *dev, int nid, bool use_dma_alloc, bool use_dma32)

Initialize a pool

Parameters

struct ttm_pool *pool

the pool to initialize

struct device *dev

device for DMA allocations and mappings

int nid

NUMA node to use for allocations

bool use_dma_alloc

true if coherent DMA alloc should be used

bool use_dma32

true if GFP_DMA32 should be used

Description

Initialize the pool and its pool types.

void **ttm_pool_fini**(struct *ttm_pool* *pool)

Cleanup a pool

Parameters

struct ttm_pool *pool

the pool to clean up

Description

Free all pages in the pool and unregister the types from the global shrinker.

int **ttm_pool_debugfs**(struct *ttm_pool* *pool, struct seq_file *m)

Debugfs dump function for a pool

Parameters

struct ttm_pool *pool

the pool to dump the information for

struct seq_file *m

seq_file to dump to

Description

Make a debugfs dump with the per pool and global information.

3.2 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.

3.2.1 GEM Initialization

Drivers that use GEM must set the DRIVER_GEM bit in the struct `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.

3.2.2 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 `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 `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 `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 `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.

3.2.3 GEM Objects Lifetime

All GEM objects are reference-counted by the GEM core. References can be acquired and release by calling `drm_gem_object_get()` and `drm_gem_object_put()` respectively.

When the last reference to a GEM object is released the GEM core calls the `struct drm_gem_object_funcs` free operation. That operation is mandatory for GEM-enabled drivers and must free the GEM object and all associated resources.

`void (*free) (struct drm_gem_object *obj);` Drivers are responsible for freeing all GEM object resources. This includes the resources created by the GEM core, which need to be released with `drm_gem_object_release()`.

3.2.4 GEM Objects Naming

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

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

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

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

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

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

3.2.5 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.

Once allocated, the fake offset value 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 `struct 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.

The VM operations is a `struct vm_operations_struct` made up of several fields, the more interesting ones being:

```
struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    void (*close)(struct vm_area_struct * area);
    vm_fault_t (*fault)(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.

For platforms without MMU the GEM core provides a helper method `drm_gem_dma_get_unmapped_area()`. The `mmap()` routines will call this to get a proposed address for the mapping.

To use `drm_gem_dma_get_unmapped_area()`, drivers must fill the struct

`struct file_operations` `get_unmapped_area` field with a pointer on `drm_gem_dma_get_unmapped_area()`.

More detailed information about `get_unmapped_area` can be found in `Documentation/admin-guide/mm/nommu-mmap.rst`

3.2.6 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).

3.2.7 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.

3.2.8 GEM Function Reference

enum `drm_gem_object_status`

bitmask of object state for fdinfo reporting

Constants

`DRM_GEM_OBJECT_RESIDENT`

object is resident in memory (ie. not unpinned)

`DRM_GEM_OBJECT_PURGEABLE`

object marked as purgeable by userspace

Description

Bitmask of status used for fdinfo memory stats, see `drm_gem_object_funcs.status` and `drm_show_fdinfo()`. Note that an object can `DRM_GEM_OBJECT_PURGEABLE` if it still active or not resident, in which case `drm_show_fdinfo()` will not account for it as purgeable. So drivers do not need to check if the buffer is idle and resident to return this bit. (Ie. userspace

can mark a buffer as purgeable even while it is still busy on the GPU.. it does not *actually* become puregeable until it becomes idle. The status gem object func does not need to consider this.)

struct **drm_gem_object_funcs**

GEM object functions

Definition:

```
struct drm_gem_object_funcs {
    void (*free)(struct drm_gem_object *obj);
    int (*open)(struct drm_gem_object *obj, struct drm_file *file);
    void (*close)(struct drm_gem_object *obj, struct drm_file *file);
    void (*print_info)(struct drm_printer *p, unsigned int indent, const
↳ struct drm_gem_object *obj);
    struct dma_buf *(*export)(struct drm_gem_object *obj, int flags);
    int (*pin)(struct drm_gem_object *obj);
    void (*unpin)(struct drm_gem_object *obj);
    struct sg_table *(*get_sg_table)(struct drm_gem_object *obj);
    int (*vmmap)(struct drm_gem_object *obj, struct iosys_map *map);
    void (*vunmap)(struct drm_gem_object *obj, struct iosys_map *map);
    int (*mmap)(struct drm_gem_object *obj, struct vm_area_struct *vma);
    int (*evict)(struct drm_gem_object *obj);
    enum drm_gem_object_status (*status)(struct drm_gem_object *obj);
    const struct vm_operations_struct *vm_ops;
};
```

Members

free

Deconstructor for `drm_gem_objects`.

This callback is mandatory.

open

Called upon GEM handle creation.

This callback is optional.

close

Called upon GEM handle release.

This callback is optional.

print_info

If driver subclasses struct `drm_gem_object`, it can implement this optional hook for printing additional driver specific info.

`drm_printf_indent()` should be used in the callback passing it the indent argument.

This callback is called from `drm_gem_print_info()`.

This callback is optional.

export

Export backing buffer as a `dma_buf`. If this is not set `drm_gem_prime_export()` is used.

This callback is optional.

pin

Pin backing buffer in memory. Used by the `drm_gem_map_attach()` helper.

This callback is optional.

unpin

Unpin backing buffer. Used by the `drm_gem_map_detach()` helper.

This callback is optional.

get_sg_table

Returns a Scatter-Gather table representation of the buffer. Used when exporting a buffer by the `drm_gem_map_dma_buf()` helper. Releasing is done by calling `dma_unmap_sg_attrs()` and `sg_free_table()` in `drm_gem_unmap_buf()`, therefore these helpers and this callback here cannot be used for sg tables pointing at driver private memory ranges.

See also `drm_prime_pages_to_sg()`.

vmap

Returns a virtual address for the buffer. Used by the `drm_gem_dmabuf_vmap()` helper.

This callback is optional.

vunmap

Releases the address previously returned by **vmap**. Used by the `drm_gem_dmabuf_vunmap()` helper.

This callback is optional.

mmap

Handle `mmap()` of the gem object, setup vma accordingly.

This callback is optional.

The callback is used by both `drm_gem_mmap_obj()` and `drm_gem_prime_mmap()`. When **mmap** is present **vm_ops** is not used, the **mmap** callback must set `vma->vm_ops` instead.

evict

Evicts gem object out from memory. Used by the `drm_gem_object_evict()` helper. Returns 0 on success, `-errno` otherwise.

This callback is optional.

status

The optional status callback can return additional object state which determines which stats the object is counted against. The callback is called under `table_lock`. Racing against object status change is “harmless”, and the callback can expect to not race against object destruction.

Called by `drm_show_memory_stats()`.

vm_ops

Virtual memory operations used with `mmap`.

This is optional but necessary for `mmap` support.

struct `drm_gem_lru`

A simple LRU helper

Definition:

```
struct drm_gem_lru {
    struct mutex *lock;
    long count;
    struct list_head list;
};
```

Members

lock

Lock protecting movement of GEM objects between LRUs. All LRUs that the object can move between should be protected by the same lock.

count

The total number of backing pages of the GEM objects in this LRU.

list

The LRU list.

Description

A helper for tracking GEM objects in a given state, to aid in driver's shrinker implementation. Tracks the count of pages for lockless shrinker.count_objects, and provides [drm_gem_lru_scan](#) for driver's shrinker.scan_objects implementation.

struct **drm_gem_object**

GEM buffer object

Definition:

```
struct drm_gem_object {
    struct kref refcount;
    unsigned handle_count;
    struct drm_device *dev;
    struct file *filp;
    struct drm_vma_offset_node vma_node;
    size_t size;
    int name;
    struct dma_buf *dma_buf;
    struct dma_buf_attachment *import_attach;
    struct dma_resv *resv;
    struct dma_resv _resv;
    struct {
        struct list_head list;
#ifdef CONFIG_LOCKDEP;
        struct lockdep_map *lock_dep_map;
#endif;
    } gpuva;
    const struct drm_gem_object_funcs *funcs;
    struct list_head lru_node;
    struct drm_gem_lru *lru;
};
```

Members

refcount

Reference count of this object

Please use `drm_gem_object_get()` to acquire and `drm_gem_object_put_locked()` or `drm_gem_object_put()` to release a reference to a GEM buffer object.

handle_count

This is the GEM file_priv handle count of this object.

Each handle also holds a reference. Note that when the handle_count drops to 0 any global names (e.g. the id in the flink namespace) will be cleared.

Protected by `drm_device.object_name_lock`.

dev

DRM dev this object belongs to.

filp

SHMEM file node used as backing storage for swappable buffer objects. GEM also supports driver private objects with driver-specific backing storage (contiguous DMA memory, special reserved blocks). In this case **filp** is NULL.

vma_node

Mapping info for this object to support mmap. Drivers are supposed to allocate the mmap offset using `drm_gem_create_mmap_offset()`. The offset itself can be retrieved using `drm_vma_node_offset_addr()`.

Memory mapping itself is handled by `drm_gem_mmap()`, which also checks that userspace is allowed to access the object.

size

Size of the object, in bytes. Immutable over the object's lifetime.

name

Global name for this object, starts at 1. 0 means unnamed. Access is covered by `drm_device.object_name_lock`. This is used by the GEM_FLINK and GEM_OPEN ioctls.

dma_buf

dma-buf associated with this GEM object.

Pointer to the dma-buf associated with this gem object (either through importing or exporting). We break the resulting reference loop when the last gem handle for this object is released.

Protected by `drm_device.object_name_lock`.

import_attach

dma-buf attachment backing this object.

Any foreign dma_buf imported as a gem object has this set to the attachment point for the device. This is invariant over the lifetime of a gem object.

The `drm_gem_object_funcs.free` callback is responsible for cleaning up the dma_buf attachment and references acquired at import time.

Note that the drm gem/prime core does not depend upon drivers setting this field any more. So for drivers where this doesn't make sense (e.g. virtual devices or a displaylink behind an usb bus) they can simply leave it as NULL.

resv

Pointer to reservation object associated with the this GEM object.

Normally (**resv** == &**_resv**) except for imported GEM objects.

_resv

A reservation object for this GEM object.

This is unused for imported GEM objects.

gpuva

Provides the list of GPU VAs attached to this GEM object.

Drivers should lock list accesses with the GEMs dma_resv lock (*drm_gem_object.resv*) or a custom lock if one is provided.

funcs

Optional GEM object functions. If this is set, it will be used instead of the corresponding *drm_driver* GEM callbacks.

New drivers should use this.

lru_node

List node in a *drm_gem_lru*.

lru

The current LRU list that the GEM object is on.

Description

This structure defines the generic parts for GEM buffer objects, which are mostly around handling mmap and userspace handles.

Buffer objects are often abbreviated to BO.

DRM_GEM_FOPS

DRM_GEM_FOPS ()

Default drm GEM file operations

Parameters

Description

This macro provides a shorthand for setting the GEM file ops in the *file_operations* structure. If all you need are the default ops, use *DEFINE_DRM_GEM_FOPS* instead.

DEFINE_DRM_GEM_FOPS

DEFINE_DRM_GEM_FOPS (name)

macro to generate file operations for GEM drivers

Parameters

name

name for the generated structure

Description

This macro autogenerates a suitable `struct file_operations` for GEM based drivers, which can be assigned to `drm_driver.fops`. Note that this structure cannot be shared between drivers, because it contains a reference to the current module using `THIS_MODULE`.

Note that the declaration is already marked as static - if you need a non-static version of this you're probably doing it wrong and will break the `THIS_MODULE` reference by accident.

```
void drm_gem_object_get(struct drm_gem_object *obj)
    acquire a GEM buffer object reference
```

Parameters

```
struct drm_gem_object *obj
    GEM buffer object
```

Description

This function acquires an additional reference to **obj**. It is illegal to call this without already holding a reference. No locks required.

```
void drm_gem_object_put(struct drm_gem_object *obj)
    drop a GEM buffer object reference
```

Parameters

```
struct drm_gem_object *obj
    GEM buffer object
```

Description

This releases a reference to **obj**.

```
bool drm_gem_object_is_shared_for_memory_stats(struct drm_gem_object *obj)
    helper for shared memory stats
```

Parameters

```
struct drm_gem_object *obj
    obj in question
```

Description

This helper should only be used for fdinfo shared memory stats to determine if a GEM object is shared.

drm_gem_gpuva_set_lock

```
drm_gem_gpuva_set_lock (obj, lock)
    Set the lock protecting accesses to the gpuva list.
```

Parameters

```
obj
    the drm_gem_object
```

```
lock
    the lock used to protect the gpuva list. The locking primitive must contain a dep_map field.
```

Description

Call this if you're not protecting access to the gpuva list with the dma-resv lock, but with a custom lock.

void **drm_gem_gpuva_init**(struct *drm_gem_object* *obj)
initialize the gpuva list of a GEM object

Parameters

struct drm_gem_object *obj
the *drm_gem_object*

Description

This initializes the *drm_gem_object*'s *drm_gpuva* list.

Calling this function is only necessary for drivers intending to support the *drm_driver_feature* DRIVER_GEM_GPUVA.

See also *drm_gem_gpuva_set_lock()*.

drm_gem_for_each_gpuva

drm_gem_for_each_gpuva (entry__, obj__)
iterator to walk over a list of gpuvas

Parameters

entry__
drm_gpuva structure to assign to in each iteration step

obj__
the *drm_gem_object* the *drm_gpuvas* to walk are associated with

Description

This iterator walks over all *drm_gpuva* structures associated with the *drm_gpuva_manager*.

drm_gem_for_each_gpuva_safe

drm_gem_for_each_gpuva_safe (entry__, next__, obj__)
iterator to safely walk over a list of gpuvas

Parameters

entry__
drm_gpuva structure to assign to in each iteration step

next__
next *drm_gpuva* to store the next step

obj__
the *drm_gem_object* the *drm_gpuvas* to walk are associated with

Description

This iterator walks over all *drm_gpuva* structures associated with the *drm_gem_object*. It is implemented with *list_for_each_entry_safe()*, hence it is safe against removal of elements.

int **drm_gem_object_init**(struct *drm_device* *dev, struct *drm_gem_object* *obj, size_t size)
initialize an allocated shmem-backed GEM object

Parameters

struct drm_device *dev
drm_device the object should be initialized for

struct drm_gem_object *obj
drm_gem_object to initialize

size_t size
object size

Description

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

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

initialize an allocated private GEM object

Parameters

struct drm_device *dev
drm_device the object should be initialized for

struct drm_gem_object *obj
drm_gem_object to initialize

size_t size
object size

Description

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

void **drm_gem_private_object_fini**(struct *drm_gem_object* *obj)

Finalize a failed drm_gem_object

Parameters

struct drm_gem_object *obj
drm_gem_object

Description

Uninitialize an already allocated GEM object when it initialized failed

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

deletes the given file-private handle

Parameters

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

u32 handle
userspace handle to delete

Description

Removes the GEM handle from the **filp** lookup table which has been added with *drm_gem_handle_create()*. If this is the last handle also cleans up linked resources like GEM names.

int **drm_gem_dumb_map_offset**(struct *drm_file* *file, struct *drm_device* *dev, u32 handle, u64 *offset)

return the fake mmap offset for a gem object

Parameters

struct drm_file *file

drm file-private structure containing the gem object

struct drm_device *dev

corresponding *drm_device*

u32 handle

gem object handle

u64 *offset

return location for the fake mmap offset

Description

This implements the *drm_driver.dumb_map_offset* kms driver callback for drivers which use gem to manage their backing storage.

Return

0 on success or a negative error code on failure.

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

create a gem handle for an object

Parameters

struct drm_file *file_priv

drm file-private structure to register the handle for

struct drm_gem_object *obj

object to register

u32 *handlep

pointer to return the created handle to the caller

Description

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

Since this publishes **obj** to userspace it must be fully set up by this point, drivers must call this last in their buffer object creation callbacks.

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

release a fake mmap offset for an object

Parameters

struct drm_gem_object *obj

obj in question

Description

This routine frees fake offsets allocated by *drm_gem_create_mmap_offset()*.

Note that `drm_gem_object_release()` already calls this function, so drivers don't have to take care of releasing the mmap offset themselves when freeing the GEM object.

int `drm_gem_create_mmap_offset_size`(struct *drm_gem_object* *obj, size_t size)
create a fake mmap offset for an object

Parameters

struct `drm_gem_object` *obj
obj in question

size_t size
the virtual size

Description

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

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

This function is idempotent and handles an already allocated mmap offset transparently. Drivers do not need to check for this case.

int `drm_gem_create_mmap_offset`(struct *drm_gem_object* *obj)
create a fake mmap offset for an object

Parameters

struct `drm_gem_object` *obj
obj in question

Description

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

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

Drivers can call `drm_gem_free_mmap_offset()` before freeing **obj** to release the fake offset again.

struct page **`drm_gem_get_pages`(struct *drm_gem_object* *obj)
helper to allocate backing pages for a GEM object from shmem

Parameters

struct `drm_gem_object` *obj
obj in question

Description

This reads the page-array of the shmem-backing storage of the given gem object. An array of pages is returned. If a page is not allocated or swapped-out, this will allocate/swap-in the required pages. Note that the whole object is covered by the page-array and pinned in memory.

Use `drm_gem_put_pages()` to release the array and unpin all pages.

This uses the GFP-mask set on the shmem-mapping (see `mapping_set_gfp_mask()`). If you require other GFP-masks, you have to do those allocations yourself.

Note that you are not allowed to change gfp-zones during runtime. That is, `shmem_read_mapping_page_gfp()` must be called with the same `gfp_zone(gfp)` as set during initialization. If you have special zone constraints, set them after `drm_gem_object_init()` via `mapping_set_gfp_mask()`. shmem-core takes care to keep pages in the required zone during swap-in.

This function is only valid on objects initialized with `drm_gem_object_init()`, but not for those initialized with `drm_gem_private_object_init()` only.

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

helper to free backing pages for a GEM object

Parameters

struct `drm_gem_object` *obj
obj in question

struct page **pages
pages to free

bool dirty
if true, pages will be marked as dirty

bool accessed
if true, the pages will be marked as accessed

int `drm_gem_objects_lookup`(struct `drm_file` *filp, void __user *bo_handles, int count, struct `drm_gem_object` ***objs_out)

look up GEM objects from an array of handles

Parameters

struct `drm_file` *filp
DRM file private data

void __user *bo_handles
user pointer to array of userspace handle

int count
size of handle array

struct `drm_gem_object` *objs_out**
returned pointer to array of `drm_gem_object` pointers

Description

Takes an array of userspace handles and returns a newly allocated array of GEM objects.

For a single handle lookup, use `drm_gem_object_lookup()`.

objs filled in with GEM object pointers. Returned GEM objects need to be released with `drm_gem_object_put()`. -ENOENT is returned on a lookup failure. 0 is returned on success.

Return

struct *drm_gem_object* ***drm_gem_object_lookup**(struct *drm_file* *filp, u32 handle)
look up a GEM object from its handle

Parameters

struct drm_file *filp
DRM file private data

u32 handle
userspace handle

Return

Description

A reference to the object named by the handle if such exists on **filp**, NULL otherwise.

If looking up an array of handles, use *drm_gem_objects_lookup()*.

long **drm_gem_dma_resv_wait**(struct *drm_file* *filep, u32 handle, bool wait_all, unsigned long timeout)

Wait on GEM object's reservation's objects shared and/or exclusive fences.

Parameters

struct drm_file *filep
DRM file private data

u32 handle
userspace handle

bool wait_all
if true, wait on all fences, else wait on just exclusive fence

unsigned long timeout
timeout value in jiffies or zero to return immediately

Return

Description

Returns -ERESTARTSYS if interrupted, 0 if the wait timed out, or greater than 0 on success.

void **drm_gem_object_release**(struct *drm_gem_object* *obj)
release GEM buffer object resources

Parameters

struct drm_gem_object *obj
GEM buffer object

Description

This releases any structures and resources used by **obj** and is the inverse of *drm_gem_object_init()*.

void **drm_gem_object_free**(struct *kref* *kref)
free a GEM object

Parameters

struct kref *kref

kref of the object to free

Description

Called after the last reference to the object has been lost.

Frees the object

void **drm_gem_vm_open**(struct vm_area_struct *vma)

vma->ops->open implementation for GEM

Parameters

struct vm_area_struct *vma

VM area structure

Description

This function implements the #vm_operations_struct open() callback for GEM drivers. This must be used together with [drm_gem_vm_close\(\)](#).

void **drm_gem_vm_close**(struct vm_area_struct *vma)

vma->ops->close implementation for GEM

Parameters

struct vm_area_struct *vma

VM area structure

Description

This function implements the #vm_operations_struct close() callback for GEM drivers. This must be used together with [drm_gem_vm_open\(\)](#).

int **drm_gem_mmap_obj**(struct [drm_gem_object](#) *obj, unsigned long obj_size, struct vm_area_struct *vma)

memory map a GEM object

Parameters

struct drm_gem_object *obj

the GEM object to map

unsigned long obj_size

the object size to be mapped, in bytes

struct vm_area_struct *vma

VMA for the area to be mapped

Description

Set up the VMA to prepare mapping of the GEM object using the GEM object's vm_ops. Depending on their requirements, GEM objects can either provide a fault handler in their vm_ops (in which case any accesses to the object will be trapped, to perform migration, GTT binding, surface register allocation, or performance monitoring), or mmap the buffer memory synchronously after calling [drm_gem_mmap_obj](#).

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

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

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

int **drm_gem_mmap**(struct file *filp, struct vm_area_struct *vma)
memory map routine for GEM objects

Parameters

struct file *filp
DRM file pointer

struct vm_area_struct *vma
VMA for the area to be mapped

Description

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

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

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

int **drm_gem_lock_reservations**(struct *drm_gem_object* **objs, int count, struct ww_acquire_ctx *acquire_ctx)
Sets up the ww context and acquires the lock on an array of GEM objects.

Parameters

struct drm_gem_object **objs
drm_gem_objects to lock

int count
Number of objects in **objs**

struct ww_acquire_ctx *acquire_ctx
struct ww_acquire_ctx that will be initialized as part of tracking this set of locked reservations.

Description

Once you've locked your reservations, you'll want to set up space for your shared fences (if applicable), submit your job, then *drm_gem_unlock_reservations()*.

void **drm_gem_lru_init**(struct *drm_gem_lru* *lru, struct mutex *lock)
initialize a LRU

Parameters

struct drm_gem_lru *lru
The LRU to initialize

struct mutex *lock
The lock protecting the LRU

void **drm_gem_lru_remove**(struct *drm_gem_object* *obj)
remove object from whatever LRU it is in

Parameters

struct drm_gem_object *obj
The GEM object to remove from current LRU

Description

If the object is currently in any LRU, remove it.

void **drm_gem_lru_move_tail_locked**(struct *drm_gem_lru* *lru, struct *drm_gem_object* *obj)
move the object to the tail of the LRU

Parameters

struct drm_gem_lru *lru
The LRU to move the object into.

struct drm_gem_object *obj
The GEM object to move into this LRU

Description

Like *drm_gem_lru_move_tail* but lru lock must be held

void **drm_gem_lru_move_tail**(struct *drm_gem_lru* *lru, struct *drm_gem_object* *obj)
move the object to the tail of the LRU

Parameters

struct drm_gem_lru *lru
The LRU to move the object into.

struct drm_gem_object *obj
The GEM object to move into this LRU

Description

If the object is already in this LRU it will be moved to the tail. Otherwise it will be removed from whichever other LRU it is in (if any) and moved into this LRU.

unsigned long **drm_gem_lru_scan**(struct *drm_gem_lru* *lru, unsigned int nr_to_scan, unsigned long *remaining, bool (*shrink)(struct *drm_gem_object* *obj))
helper to implement shrinker.scan_objects

Parameters

struct drm_gem_lru *lru
The LRU to scan

unsigned int nr_to_scan
The number of pages to try to reclaim

unsigned long *remaining
The number of pages left to reclaim, should be initialized by caller

bool (*shrink)(struct drm_gem_object *obj)
Callback to try to shrink/reclaim the object.

Description

If the shrink callback succeeds, it is expected that the driver move the object out of this LRU. If the LRU possibly contain active buffers, it is the responsibility of the shrink callback to check for this (ie. `drm_resv_test_signaled()`) or if necessary block until the buffer becomes idle.

int **drm_gem_evict**(struct *drm_gem_object* *obj)
 helper to evict backing pages for a GEM object

Parameters

struct drm_gem_object *obj
 obj in question

3.2.9 GEM DMA Helper Functions Reference

The DRM GEM/DMA helpers are a means to provide buffer objects that are presented to the device as a contiguous chunk of memory. This is useful for devices that do not support scatter-gather DMA (either directly or by using an intimately attached IOMMU).

For devices that access the memory bus through an (external) IOMMU then the buffer objects are allocated using a traditional page-based allocator and may be scattered through physical memory. However they are contiguous in the IOVA space so appear contiguous to devices using them.

For other devices then the helpers rely on CMA to provide buffer objects that are physically contiguous in memory.

For GEM callback helpers in struct *drm_gem_object* functions, see likewise named functions with an `_object_infix` (e.g., `drm_gem_dma_object_vmap()` wraps *drm_gem_dma_vmap()*). These helpers perform the necessary type conversion.

struct drm_gem_dma_object
 GEM object backed by DMA memory allocations

Definition:

```
struct drm_gem_dma_object {
    struct drm_gem_object base;
    dma_addr_t dma_addr;
    struct sg_table *sgt;
    void *vaddr;
    bool map_noncoherent;
};
```

Members

base
 base GEM object

dma_addr
 DMA address of the backing memory

sgt
 scatter/gather table for imported PRIME buffers. The table can have more than one entry but they are guaranteed to have contiguous DMA addresses.

vaddr

kernel virtual address of the backing memory

map_noncoherent

if true, the GEM object is backed by non-coherent memory

void **drm_gem_dma_object_free**(struct *drm_gem_object* *obj)

GEM object function for *drm_gem_dma_free()*

Parameters

struct drm_gem_object *obj

GEM object to free

Description

This function wraps *drm_gem_dma_free_object()*. Drivers that employ the DMA helpers should use it as their *drm_gem_object_funcs.free* handler.

void **drm_gem_dma_object_print_info**(struct *drm_printer* *p, unsigned int indent, const struct *drm_gem_object* *obj)

Print *drm_gem_dma_object* info for debugfs

Parameters

struct drm_printer *p

DRM printer

unsigned int indent

Tab indentation level

const struct drm_gem_object *obj

GEM object

Description

This function wraps *drm_gem_dma_print_info()*. Drivers that employ the DMA helpers should use this function as their *drm_gem_object_funcs.print_info* handler.

struct sg_table ***drm_gem_dma_object_get_sg_table**(struct *drm_gem_object* *obj)

GEM object function for *drm_gem_dma_get_sg_table()*

Parameters

struct drm_gem_object *obj

GEM object

Description

This function wraps *drm_gem_dma_get_sg_table()*. Drivers that employ the DMA helpers should use it as their *drm_gem_object_funcs.get_sg_table* handler.

Return

A pointer to the scatter/gather table of pinned pages or NULL on failure.

int **drm_gem_dma_object_mmap**(struct *drm_gem_object* *obj, struct vm_area_struct *vma)

GEM object function for *drm_gem_dma_mmap()*

Parameters

struct drm_gem_object *obj

GEM object

struct vm_area_struct *vma

VMA for the area to be mapped

Description

This function wraps [drm_gem_dma_mmap\(\)](#). Drivers that employ the dma helpers should use it as their [drm_gem_object_funcs.mmap](#) handler.

Return

0 on success or a negative error code on failure.

DRM_GEM_DMA_DRIVER_OPS_WITH_DUMB_CREATE

DRM_GEM_DMA_DRIVER_OPS_WITH_DUMB_CREATE (dumb_create_func)

DMA GEM driver operations

Parameters

dumb_create_func

callback function for .dumb_create

Description

This macro provides a shortcut for setting the default GEM operations in the [drm_driver](#) structure.

This macro is a variant of `DRM_GEM_DMA_DRIVER_OPS` for drivers that override the default implementation of `struct rm_driver.dumb_create`. Use `DRM_GEM_DMA_DRIVER_OPS` if possible. Drivers that require a virtual address on imported buffers should use [DRM_GEM_DMA_DRIVER_OPS_VMAP_WITH_DUMB_CREATE\(\)](#) instead.

DRM_GEM_DMA_DRIVER_OPS

DRM_GEM_DMA_DRIVER_OPS ()

DMA GEM driver operations

Parameters

Description

This macro provides a shortcut for setting the default GEM operations in the [drm_driver](#) structure.

Drivers that come with their own implementation of [struct drm_driver.dumb_create](#) should use [DRM_GEM_DMA_DRIVER_OPS_WITH_DUMB_CREATE\(\)](#) instead. Use `DRM_GEM_DMA_DRIVER_OPS` if possible. Drivers that require a virtual address on imported buffers should use `DRM_GEM_DMA_DRIVER_OPS_VMAP` instead.

DRM_GEM_DMA_DRIVER_OPS_VMAP_WITH_DUMB_CREATE

DRM_GEM_DMA_DRIVER_OPS_VMAP_WITH_DUMB_CREATE (dumb_create_func)

DMA GEM driver operations ensuring a virtual address on the buffer

Parameters

dumb_create_func

callback function for .dumb_create

Description

This macro provides a shortcut for setting the default GEM operations in the *drm_driver* structure for drivers that need the virtual address also on imported buffers.

This macro is a variant of `DRM_GEM_DMA_DRIVER_OPS_VMAP` for drivers that override the default implementation of *struct drm_driver.dumb_create*. Use `DRM_GEM_DMA_DRIVER_OPS_VMAP` if possible. Drivers that do not require a virtual address on imported buffers should use *DRM_GEM_DMA_DRIVER_OPS_WITH_DUMB_CREATE()* instead.

DRM_GEM_DMA_DRIVER_OPS_VMAP

`DRM_GEM_DMA_DRIVER_OPS_VMAP ()`

DMA GEM driver operations ensuring a virtual address on the buffer

Parameters

Description

This macro provides a shortcut for setting the default GEM operations in the *drm_driver* structure for drivers that need the virtual address also on imported buffers.

Drivers that come with their own implementation of *struct drm_driver.dumb_create* should use *DRM_GEM_DMA_DRIVER_OPS_VMAP_WITH_DUMB_CREATE()* instead. Use `DRM_GEM_DMA_DRIVER_OPS_VMAP` if possible. Drivers that do not require a virtual address on imported buffers should use `DRM_GEM_DMA_DRIVER_OPS` instead.

DEFINE_DRM_GEM_DMA_FOPS

`DEFINE_DRM_GEM_DMA_FOPS (name)`

macro to generate file operations for DMA drivers

Parameters

name

name for the generated structure

Description

This macro autogenerates a suitable *struct file_operations* for DMA based drivers, which can be assigned to *drm_driver.fops*. Note that this structure cannot be shared between drivers, because it contains a reference to the current module using `THIS_MODULE`.

Note that the declaration is already marked as static - if you need a non-static version of this you're probably doing it wrong and will break the `THIS_MODULE` reference by accident.

struct drm_gem_dma_object ***drm_gem_dma_create**(*struct drm_device* *drm, *size_t* size)

allocate an object with the given size

Parameters

struct drm_device *drm

DRM device

size_t size

size of the object to allocate

Description

This function creates a DMA GEM object and allocates memory as backing store. The allocated memory will occupy a contiguous chunk of bus address space.

For devices that are directly connected to the memory bus then the allocated memory will be physically contiguous. For devices that access through an IOMMU, then the allocated memory is not expected to be physically contiguous because having contiguous IOVAs is sufficient to meet a devices DMA requirements.

Return

A *struct [drm_gem_dma_object](#)* * on success or an ERR_PTR()-encoded negative error code on failure.

```
void drm_gem_dma_free(struct drm_gem_dma_object *dma_obj)
    free resources associated with a DMA GEM object
```

Parameters

```
struct drm_gem_dma_object *dma_obj
    DMA GEM object to free
```

Description

This function frees the backing memory of the DMA GEM object, cleans up the GEM object state and frees the memory used to store the object itself. If the buffer is imported and the virtual address is set, it is released.

```
int drm_gem_dma_dumb_create_internal(struct drm_file *file_priv, struct drm_device *drm,
                                     struct drm_mode_create_dumb *args)
    create a dumb buffer object
```

Parameters

```
struct drm_file *file_priv
    DRM file-private structure to create the dumb buffer for

struct drm_device *drm
    DRM device

struct drm_mode_create_dumb *args
    IOCTL data
```

Description

This aligns the pitch and size arguments to the minimum required. This is an internal helper that can be wrapped by a driver to account for hardware with more specific alignment requirements. It should not be used directly as their *drm_driver.dumb_create* callback.

Return

0 on success or a negative error code on failure.

```
int drm_gem_dma_dumb_create(struct drm_file *file_priv, struct drm_device *drm, struct
                             drm_mode_create_dumb *args)
    create a dumb buffer object
```

Parameters

struct drm_file *file_priv

DRM file-private structure to create the dumb buffer for

struct drm_device *drm

DRM device

struct drm_mode_create_dumb *args

IOCTL data

Description

This function computes the pitch of the dumb buffer and rounds it up to an integer number of bytes per pixel. Drivers for hardware that doesn't have any additional restrictions on the pitch can directly use this function as their *drm_driver.dumb_create* callback.

For hardware with additional restrictions, drivers can adjust the fields set up by userspace and pass the IOCTL data along to the *drm_gem_dma_dumb_create_internal()* function.

Return

0 on success or a negative error code on failure.

unsigned long **drm_gem_dma_get_unmapped_area**(struct file *filp, unsigned long addr,
unsigned long len, unsigned long pgoff,
unsigned long flags)

propose address for mapping in noMMU cases

Parameters

struct file *filp

file object

unsigned long addr

memory address

unsigned long len

buffer size

unsigned long pgoff

page offset

unsigned long flags

memory flags

Description

This function is used in noMMU platforms to propose address mapping for a given buffer. It's intended to be used as a direct handler for the struct file_operations.get_unmapped_area operation.

Return

mapping address on success or a negative error code on failure.

void **drm_gem_dma_print_info**(const struct *drm_gem_dma_object* *dma_obj, struct
drm_printer *p, unsigned int indent)

Print *drm_gem_dma_object* info for debugfs

Parameters

```
const struct drm_gem_dma_object *dma_obj
```

DMA GEM object

```
struct drm_printer *p
```

DRM printer

```
unsigned int indent
```

Tab indentation level

Description

This function prints `dma_addr` and `vaddr` for use in e.g. `debugfs` output.

```
struct sg_table *drm_gem_dma_get_sg_table(struct drm_gem_dma_object *dma_obj)
```

provide a scatter/gather table of pinned pages for a DMA GEM object

Parameters

```
struct drm_gem_dma_object *dma_obj
```

DMA GEM object

Description

This function exports a scatter/gather table by calling the standard DMA mapping API.

Return

A pointer to the scatter/gather table of pinned pages or `NULL` on failure.

```
struct drm_gem_object *drm_gem_dma_prime_import_sg_table(struct drm_device *dev,  
                                                         struct dma_buf_attachment  
                                                         *attach, struct sg_table *sgt)
```

produce a DMA GEM object from another driver's scatter/gather table of pinned pages

Parameters

```
struct drm_device *dev
```

device to import into

```
struct dma_buf_attachment *attach
```

DMA-BUF attachment

```
struct sg_table *sgt
```

scatter/gather table of pinned pages

Description

This function imports a scatter/gather table exported via DMA-BUF by another driver. Imported buffers must be physically contiguous in memory (i.e. the scatter/gather table must contain a single entry). Drivers that use the DMA helpers should set this as their *drm_driver.gem_prime_import_sg_table* callback.

Return

A pointer to a newly created GEM object or an `ERR_PTR`-encoded negative error code on failure.

```
int drm_gem_dma_vmap(struct drm_gem_dma_object *dma_obj, struct iosys_map *map)
```

map a DMA GEM object into the kernel's virtual address space

Parameters

struct *drm_gem_dma_object* *dma_obj

DMA GEM object

struct *iosys_map* *map

Returns the kernel virtual address of the DMA GEM object's backing store.

Description

This function maps a buffer into the kernel's virtual address space. Since the DMA buffers are already mapped into the kernel virtual address space this simply returns the cached virtual address.

Return

0 on success, or a negative error code otherwise.

int **drm_gem_dma_mmap**(struct *drm_gem_dma_object* *dma_obj, struct *vm_area_struct* *vma)

memory-map an exported DMA GEM object

Parameters

struct *drm_gem_dma_object* *dma_obj

DMA GEM object

struct *vm_area_struct* *vma

VMA for the area to be mapped

Description

This function maps a buffer into a userspace process's address space. In addition to the usual GEM VMA setup it immediately faults in the entire object instead of using on-demand faulting.

Return

0 on success or a negative error code on failure.

struct *drm_gem_object* ***drm_gem_dma_prime_import_sg_table_vmap**(struct *drm_device* *dev,
struct
dma_buf_attachment
*attach, struct *sg_table*
*sgt)

PRIME import another driver's scatter/gather table and get the virtual address of the buffer

Parameters

struct *drm_device* *dev

DRM device

struct *dma_buf_attachment* *attach

DMA-BUF attachment

struct *sg_table* *sgt

Scatter/gather table of pinned pages

Description

This function imports a scatter/gather table using *drm_gem_dma_prime_import_sg_table()* and uses *dma_buf_vmap()* to get the kernel virtual address. This ensures that a DMA GEM object always has its virtual address set. This address is released when the object is freed.

This function can be used as the `drm_driver.gem_prime_import_sg_table` callback. The `DRM_GEM_DMA_DRIVER_OPS_VMAP` macro provides a shortcut to set the necessary DRM driver operations.

Return

A pointer to a newly created GEM object or an ERR_PTR-encoded negative error code on failure.

3.2.10 GEM SHMEM Helper Function Reference

This library provides helpers for GEM objects backed by shmem buffers allocated using anonymous pageable memory.

Functions that operate on the GEM object receive struct `drm_gem_shmem_object`. For GEM callback helpers in struct `drm_gem_object` functions, see likewise named functions with an `_object_infix` (e.g., `drm_gem_shmem_object_vmap()` wraps `drm_gem_shmem_vmap()`). These helpers perform the necessary type conversion.

struct **drm_gem_shmem_object**

GEM object backed by shmem

Definition:

```
struct drm_gem_shmem_object {
    struct drm_gem_object base;
    struct page **pages;
    unsigned int pages_use_count;
    int madv;
    struct list_head madv_list;
    struct sg_table *sgt;
    void *vaddr;
    unsigned int vmap_use_count;
    bool pages_mark_dirty_on_put : 1;
    bool pages_mark_accessed_on_put : 1;
    bool map_wc : 1;
};
```

Members

base

Base GEM object

pages

Page table

pages_use_count

Reference count on the pages table. The pages are put when the count reaches zero.

madv

State for madvise

0 is active/inuse. A negative value is the object is purged. Positive values are driver specific and not used by the helpers.

madv_list

List entry for madvise tracking

Typically used by drivers to track purgeable objects

sgt

Scatter/gather table for imported PRIME buffers

vaddr

Kernel virtual address of the backing memory

vmap_use_count

Reference count on the virtual address. The address are un-mapped when the count reaches zero.

pages_mark_dirty_on_put

Mark pages as dirty when they are put.

pages_mark_accessed_on_put

Mark pages as accessed when they are put.

map_wc

map object write-combined (instead of using shmem defaults).

void **drm_gem_shmem_object_free**(struct *drm_gem_object* *obj)

GEM object function for *drm_gem_shmem_free()*

Parameters

struct drm_gem_object *obj

GEM object to free

Description

This function wraps *drm_gem_shmem_free()*. Drivers that employ the shmem helpers should use it as their *drm_gem_object_funcs.free* handler.

void **drm_gem_shmem_object_print_info**(struct *drm_printer* *p, unsigned int indent, const struct *drm_gem_object* *obj)

Print *drm_gem_shmem_object* info for debugfs

Parameters

struct drm_printer *p

DRM printer

unsigned int indent

Tab indentation level

const struct drm_gem_object *obj

GEM object

Description

This function wraps *drm_gem_shmem_print_info()*. Drivers that employ the shmem helpers should use this function as their *drm_gem_object_funcs.print_info* handler.

int **drm_gem_shmem_object_pin**(struct *drm_gem_object* *obj)

GEM object function for *drm_gem_shmem_pin()*

Parameters

struct drm_gem_object *obj

GEM object

Description

This function wraps `drm_gem_shmem_pin()`. Drivers that employ the shmem helpers should use it as their `drm_gem_object_funcs.pin` handler.

```
void drm_gem_shmem_object_unpin(struct drm_gem_object *obj)
    GEM object function for drm_gem_shmem_unpin()
```

Parameters

```
struct drm_gem_object *obj
    GEM object
```

Description

This function wraps `drm_gem_shmem_unpin()`. Drivers that employ the shmem helpers should use it as their `drm_gem_object_funcs.unpin` handler.

```
struct sg_table *drm_gem_shmem_object_get_sg_table(struct drm_gem_object *obj)
    GEM object function for drm_gem_shmem_get_sg_table()
```

Parameters

```
struct drm_gem_object *obj
    GEM object
```

Description

This function wraps `drm_gem_shmem_get_sg_table()`. Drivers that employ the shmem helpers should use it as their `drm_gem_object_funcs.get_sg_table` handler.

Return

A pointer to the scatter/gather table of pinned pages or error pointer on failure.

```
int drm_gem_shmem_object_mmap(struct drm_gem_object *obj, struct vm_area_struct *vma)
    GEM object function for drm_gem_shmem_mmap()
```

Parameters

```
struct drm_gem_object *obj
    GEM object
```

```
struct vm_area_struct *vma
    VMA for the area to be mapped
```

Description

This function wraps `drm_gem_shmem_mmap()`. Drivers that employ the shmem helpers should use it as their `drm_gem_object_funcs.mmap` handler.

Return

0 on success or a negative error code on failure.

DRM_GEM_SHMEM_DRIVER_OPS

```
DRM_GEM_SHMEM_DRIVER_OPS ( )
```

Default shmem GEM operations

Parameters

Description

This macro provides a shortcut for setting the shmem GEM operations in the *drm_driver* structure.

struct *drm_gem_shmem_object* ***drm_gem_shmem_create**(struct *drm_device* *dev, size_t size)
Allocate an object with the given size

Parameters

struct *drm_device* *dev
DRM device

size_t size
Size of the object to allocate

Description

This function creates a shmem GEM object.

Return

A struct *drm_gem_shmem_object* * on success or an ERR_PTR()-encoded negative error code on failure.

void **drm_gem_shmem_free**(struct *drm_gem_shmem_object* *shmem)
Free resources associated with a shmem GEM object

Parameters

struct *drm_gem_shmem_object* *shmem
shmem GEM object to free

Description

This function cleans up the GEM object state and frees the memory used to store the object itself.

int **drm_gem_shmem_pin**(struct *drm_gem_shmem_object* *shmem)
Pin backing pages for a shmem GEM object

Parameters

struct *drm_gem_shmem_object* *shmem
shmem GEM object

Description

This function makes sure the backing pages are pinned in memory while the buffer is exported.

Return

0 on success or a negative error code on failure.

void **drm_gem_shmem_unpin**(struct *drm_gem_shmem_object* *shmem)
Unpin backing pages for a shmem GEM object

Parameters

struct *drm_gem_shmem_object* *shmem
shmem GEM object

Description

This function removes the requirement that the backing pages are pinned in memory.

```
int drm_gem_shmem_dumb_create(struct drm_file *file, struct drm_device *dev, struct  
                             drm_mode_create_dumb *args)
```

Create a dumb shmem buffer object

Parameters

struct *drm_file* *file

DRM file structure to create the dumb buffer for

struct *drm_device* *dev

DRM device

struct *drm_mode_create_dumb* *args

IOCTL data

Description

This function computes the pitch of the dumb buffer and rounds it up to an integer number of bytes per pixel. Drivers for hardware that doesn't have any additional restrictions on the pitch can directly use this function as their *drm_driver.dumb_create* callback.

For hardware with additional restrictions, drivers can adjust the fields set up by userspace before calling into this function.

Return

0 on success or a negative error code on failure.

```
int drm_gem_shmem_mmap(struct drm_gem_shmem_object *shmem, struct vm_area_struct  
                       *vma)
```

Memory-map a shmem GEM object

Parameters

struct *drm_gem_shmem_object* *shmem

shmem GEM object

struct *vm_area_struct* *vma

VMA for the area to be mapped

Description

This function implements an augmented version of the GEM DRM file mmap operation for shmem objects.

Return

0 on success or a negative error code on failure.

```
void drm_gem_shmem_print_info(const struct drm_gem_shmem_object *shmem, struct  
                             drm_printer *p, unsigned int indent)
```

Print *drm_gem_shmem_object* info for debugfs

Parameters

const struct *drm_gem_shmem_object* *shmem

shmem GEM object

struct drm_printer *p

DRM printer

unsigned int indent

Tab indentation level

struct sg_table *drm_gem_shmem_get_sg_table(struct *drm_gem_shmem_object* *shmem)

Provide a scatter/gather table of pinned pages for a shmem GEM object

Parameters

struct drm_gem_shmem_object *shmem

shmem GEM object

Description

This function exports a scatter/gather table suitable for PRIME usage by calling the standard DMA mapping API.

Drivers who need to acquire a scatter/gather table for objects need to call *drm_gem_shmem_get_pages_sgt()* instead.

Return

A pointer to the scatter/gather table of pinned pages or error pointer on failure.

struct sg_table *drm_gem_shmem_get_pages_sgt(struct *drm_gem_shmem_object* *shmem)

Pin pages, dma map them, and return a scatter/gather table for a shmem GEM object.

Parameters

struct drm_gem_shmem_object *shmem

shmem GEM object

Description

This function returns a scatter/gather table suitable for driver usage. If the sg table doesn't exist, the pages are pinned, dma-mapped, and a sg table created.

This is the main function for drivers to get at backing storage, and it hides any difference between dma-buf imported and natively allocated objects. *drm_gem_shmem_get_sg_table()* should not be directly called by drivers.

Return

A pointer to the scatter/gather table of pinned pages or errno on failure.

struct *drm_gem_object* *drm_gem_shmem_prime_import_sg_table(struct *drm_device* *dev,
struct
dma_buf_attachment
*attach, struct sg_table
*sgt)

Produce a shmem GEM object from another driver's scatter/gather table of pinned pages

Parameters

struct drm_device *dev

Device to import into

struct dma_buf_attachment *attach

DMA-BUF attachment

```
struct sg_table *sgt
```

Scatter/gather table of pinned pages

Description

This function imports a scatter/gather table exported via DMA-BUF by another driver. Drivers that use the shmem helpers should set this as their `drm_driver.gem_prime_import_sg_table` callback.

Return

A pointer to a newly created GEM object or an ERR_PTR-encoded negative error code on failure.

3.2.11 GEM VRAM Helper Functions Reference

This library provides `struct drm_gem_vram_object` (GEM VRAM), a GEM buffer object that is backed by video RAM (VRAM). It can be used for framebuffer devices with dedicated memory.

The data structure `struct drm_vram_mm` and its helpers implement a memory manager for simple framebuffer devices with dedicated video memory. GEM VRAM buffer objects are either placed in the video memory or remain evicted to system memory.

With the GEM interface userspace applications create, manage and destroy graphics buffers, such as an on-screen framebuffer. GEM does not provide an implementation of these interfaces. It's up to the DRM driver to provide an implementation that suits the hardware. If the hardware device contains dedicated video memory, the DRM driver can use the VRAM helper library. Each active buffer object is stored in video RAM. Active buffers are used for drawing the current frame, typically something like the frame's scanout buffer or the cursor image. If there's no more space left in VRAM, inactive GEM objects can be moved to system memory.

To initialize the VRAM helper library call `drmm_vram_helper_init()`. The function allocates and initializes an instance of `struct drm_vram_mm` in `struct drm_device.vram_mm`. Use `DRM_GEM_VRAM_DRIVER` to initialize `struct drm_driver` and `DRM_VRAM_MM_FILE_OPERATIONS` to initialize `struct file_operations`; as illustrated below.

```
struct file_operations fops = {
    .owner = THIS_MODULE,
    DRM_VRAM_MM_FILE_OPERATION
};
struct drm_driver drv = {
    .driver_feature = DRM_ ... ,
    .fops = &fops,
    DRM_GEM_VRAM_DRIVER
};

int init_drm_driver()
{
    struct drm_device *dev;
    uint64_t vram_base;
    unsigned long vram_size;
    int ret;

    // setup device, vram base and size
    // ...
}
```

```
    ret = drmm_vram_helper_init(dev, vram_base, vram_size);
    if (ret)
        return ret;
    return 0;
}
```

This creates an instance of `struct drm_vram_mm`, exports DRM userspace interfaces for GEM buffer management and initializes file operations to allow for accessing created GEM buffers. With this setup, the DRM driver manages an area of video RAM with VRAM MM and provides GEM VRAM objects to userspace.

You don't have to clean up the instance of VRAM MM. `drmm_vram_helper_init()` is a managed interface that installs a clean-up handler to run during the DRM device's release.

For drawing or scanout operations, resp. buffer objects have to be pinned in video RAM. Call `drm_gem_vram_pin()` with `DRM_GEM_VRAM_PL_FLAG_VRAM` or `DRM_GEM_VRAM_PL_FLAG_SYSTEM` to pin a buffer object in video RAM or system memory. Call `drm_gem_vram_unpin()` to release the pinned object afterwards.

A buffer object that is pinned in video RAM has a fixed address within that memory region. Call `drm_gem_vram_offset()` to retrieve this value. Typically it's used to program the hardware's scanout engine for framebuffers, set the cursor overlay's image for a mouse cursor, or use it as input to the hardware's drawing engine.

To access a buffer object's memory from the DRM driver, call `drm_gem_vram_vmap()`. It maps the buffer into kernel address space and returns the memory address. Use `drm_gem_vram_vunmap()` to release the mapping.

struct **drm_gem_vram_object**
GEM object backed by VRAM

Definition:

```
struct drm_gem_vram_object {
    struct ttm_buffer_object bo;
    struct iosys_map map;
    unsigned int vmap_use_count;
    struct ttm_placement placement;
    struct ttm_place placements[2];
};
```

Members

bo
TTM buffer object

map
Mapping information for **bo**

vmap_use_count
Reference count on the virtual address. The address are un-mapped when the count reaches zero.

placement
TTM placement information. Supported placements are `TTM_PL_VRAM` and `TTM_PL_SYSTEM`

placements

TTM placement information.

Description

The type *struct drm_gem_vram_object* represents a GEM object that is backed by VRAM. It can be used for simple framebuffer devices with dedicated memory. The buffer object can be evicted to system memory if video memory becomes scarce.

GEM VRAM objects perform reference counting for pin and mapping operations. So a buffer object that has been pinned N times with *drm_gem_vram_pin()* must be unpinned N times with *drm_gem_vram_unpin()*. The same applies to pairs of *drm_gem_vram_kmap()* and *drm_gem_vram_kunmap()*, as well as pairs of *drm_gem_vram_vmap()* and *drm_gem_vram_vunmap()*.

```
struct drm_gem_vram_object *drm_gem_vram_of_bo(struct ttm_buffer_object *bo)
```

Returns the container of type *struct drm_gem_vram_object* for field bo.

Parameters

struct ttm_buffer_object *bo
the VRAM buffer object

Return

The containing GEM VRAM object

```
struct drm_gem_vram_object *drm_gem_vram_of_gem(struct drm_gem_object *gem)
```

Returns the container of type *struct drm_gem_vram_object* for field gem.

Parameters

struct drm_gem_object *gem
the GEM object

Return

The containing GEM VRAM object

DRM_GEM_VRAM_PLANE_HELPER_FUNCS

```
DRM_GEM_VRAM_PLANE_HELPER_FUNCS ( )
```

Initializes *struct drm_plane_helper_funcs* for VRAM handling

Parameters**Description**

Drivers may use GEM BOs as VRAM helpers for the framebuffer memory. This macro initializes *struct drm_plane_helper_funcs* to use the respective helper functions.

DRM_GEM_VRAM_DRIVER

```
DRM_GEM_VRAM_DRIVER ( )
```

default callback functions for *struct drm_driver*

Parameters**Description**

Drivers that use VRAM MM and GEM VRAM can use this macro to initialize *struct drm_driver* with default functions.

struct **drm_vram_mm**

An instance of VRAM MM

Definition:

```
struct drm_vram_mm {
    uint64_t vram_base;
    size_t vram_size;
    struct ttm_device bdev;
};
```

Members

vram_base

Base address of the managed video memory

vram_size

Size of the managed video memory in bytes

bdev

The TTM BO device.

Description

The fields *struct drm_vram_mm.vram_base* and *struct drm_vram_mm.vram_size* are managed by VRAM MM, but are available for public read access. Use the field *struct drm_vram_mm.bdev* to access the TTM BO device.

struct *drm_vram_mm* ***drm_vram_mm_of_bdev**(struct *ttm_device* *bdev)

Returns the container of type *struct ttm_device* for field bdev.

Parameters

struct *ttm_device* ***bdev**

the TTM BO device

Return

The containing instance of *struct drm_vram_mm*

struct *drm_gem_vram_object* ***drm_gem_vram_create**(struct *drm_device* *dev, size_t size,
unsigned long pg_align)

Creates a VRAM-backed GEM object

Parameters

struct *drm_device* ***dev**

the DRM device

size_t **size**

the buffer size in bytes

unsigned long **pg_align**

the buffer's alignment in multiples of the page size

Description

GEM objects are allocated by calling `struct drm_driver.gem_create_object`, if set. Otherwise `kzalloc()` will be used. Drivers can set their own GEM object functions in `struct drm_driver.gem_create_object`. If no functions are set, the new GEM object will use the default functions from GEM VRAM helpers.

Return

A new instance of `struct drm_gem_vram_object` on success, or an `ERR_PTR()`-encoded error code otherwise.

`void drm_gem_vram_put(struct drm_gem_vram_object *gbo)`
Releases a reference to a VRAM-backed GEM object

Parameters

`struct drm_gem_vram_object *gbo`
the GEM VRAM object

Description

See `ttm_bo_put()` for more information.

`s64 drm_gem_vram_offset(struct drm_gem_vram_object *gbo)`
Returns a GEM VRAM object's offset in video memory

Parameters

`struct drm_gem_vram_object *gbo`
the GEM VRAM object

Description

This function returns the buffer object's offset in the device's video memory. The buffer object has to be pinned to `TTM_PL_VRAM`.

Return

The buffer object's offset in video memory on success, or a negative `errno` code otherwise.

`int drm_gem_vram_pin(struct drm_gem_vram_object *gbo, unsigned long pl_flag)`
Pins a GEM VRAM object in a region.

Parameters

`struct drm_gem_vram_object *gbo`
the GEM VRAM object

`unsigned long pl_flag`
a bitmask of possible memory regions

Description

Pinning a buffer object ensures that it is not evicted from a memory region. A pinned buffer object has to be unpinned before it can be pinned to another region. If the `pl_flag` argument is 0, the buffer is pinned at its current location (video RAM or system memory).

Small buffer objects, such as cursor images, can lead to memory fragmentation if they are pinned in the middle of video RAM. This is especially a problem on devices with only a small amount of video RAM. Fragmentation can prevent the primary framebuffer from fitting in, even though there's enough memory overall. The modifier `DRM_GEM_VRAM_PL_FLAG_TOPDOWN`

marks the buffer object to be pinned at the high end of the memory region to avoid fragmentation.

Return

0 on success, or a negative error code otherwise.

int **drm_gem_vram_unpin**(struct *drm_gem_vram_object* *gbo)

Unpins a GEM VRAM object

Parameters

struct drm_gem_vram_object *gbo

the GEM VRAM object

Return

0 on success, or a negative error code otherwise.

int **drm_gem_vram_vmap**(struct *drm_gem_vram_object* *gbo, struct iosys_map *map)

Pins and maps a GEM VRAM object into kernel address space

Parameters

struct drm_gem_vram_object *gbo

The GEM VRAM object to map

struct iosys_map *map

Returns the kernel virtual address of the VRAM GEM object's backing store.

Description

The vmap function pins a GEM VRAM object to its current location, either system or video memory, and maps its buffer into kernel address space. As pinned object cannot be relocated, you should avoid pinning objects permanently. Call *drm_gem_vram_vunmap()* with the returned address to unmap and unpin the GEM VRAM object.

Return

0 on success, or a negative error code otherwise.

void **drm_gem_vram_vunmap**(struct *drm_gem_vram_object* *gbo, struct iosys_map *map)

Unmaps and unpins a GEM VRAM object

Parameters

struct drm_gem_vram_object *gbo

The GEM VRAM object to unmap

struct iosys_map *map

Kernel virtual address where the VRAM GEM object was mapped

Description

A call to *drm_gem_vram_vunmap()* unmaps and unpins a GEM VRAM buffer. See the documentation for *drm_gem_vram_vmap()* for more information.

int **drm_gem_vram_fill_create_dumb**(struct *drm_file* *file, struct *drm_device* *dev, unsigned long pg_align, unsigned long pitch_align, struct *drm_mode_create_dumb* *args)

Helper for implementing *struct drm_driver.dumb_create*

Parameters

struct drm_file *file
the DRM file

struct drm_device *dev
the DRM device

unsigned long pg_align
the buffer's alignment in multiples of the page size

unsigned long pitch_align
the scanline's alignment in powers of 2

struct drm_mode_create_dumb *args
the arguments as provided to *struct drm_driver.dumb_create*

Description

This helper function fills *struct drm_mode_create_dumb*, which is used by *struct drm_driver.dumb_create*. Implementations of this interface should forwards their arguments to this helper, plus the driver-specific parameters.

Return

0 on success, or a negative error code otherwise.

int **drm_gem_vram_driver_dumb_create**(struct *drm_file* *file, struct *drm_device* *dev, struct *drm_mode_create_dumb* *args)

Implements *struct drm_driver.dumb_create*

Parameters

struct drm_file *file
the DRM file

struct drm_device *dev
the DRM device

struct drm_mode_create_dumb *args
the arguments as provided to *struct drm_driver.dumb_create*

Description

This function requires the driver to use **drm_device.vram_mm** for its instance of VRAM MM.

Return

0 on success, or a negative error code otherwise.

int **drm_gem_vram_plane_helper_prepare_fb**(struct *drm_plane* *plane, struct *drm_plane_state* *new_state)

- Implements *struct drm_plane_helper_funcs.prepare_fb*

Parameters

struct drm_plane *plane
a DRM plane

struct drm_plane_state *new_state
the plane's new state

Description

During plane updates, this function sets the plane's fence and pins the GEM VRAM objects of the plane's new framebuffer to VRAM. Call `drm_gem_vram_plane_helper_cleanup_fb()` to unpin them.

Return

0 on success, or a negative errno code otherwise.

void **drm_gem_vram_plane_helper_cleanup_fb**(struct *drm_plane* *plane, struct *drm_plane_state* *old_state)

- Implements *struct drm_plane_helper_funcs*.cleanup_fb

Parameters

struct drm_plane *plane
a DRM plane

struct drm_plane_state *old_state
the plane's old state

Description

During plane updates, this function unpins the GEM VRAM objects of the plane's old framebuffer from VRAM. Complements `drm_gem_vram_plane_helper_prepare_fb()`.

int **drm_gem_vram_simple_display_pipe_prepare_fb**(struct *drm_simple_display_pipe* *pipe, struct *drm_plane_state* *new_state)

- Implements *struct drm_simple_display_pipe_funcs*.prepare_fb

Parameters

struct drm_simple_display_pipe *pipe
a simple display pipe

struct drm_plane_state *new_state
the plane's new state

Description

During plane updates, this function pins the GEM VRAM objects of the plane's new framebuffer to VRAM. Call `drm_gem_vram_simple_display_pipe_cleanup_fb()` to unpin them.

Return

0 on success, or a negative errno code otherwise.

void **drm_gem_vram_simple_display_pipe_cleanup_fb**(struct *drm_simple_display_pipe* *pipe, struct *drm_plane_state* *old_state)

- Implements *struct drm_simple_display_pipe_funcs*.cleanup_fb

Parameters

struct drm_simple_display_pipe *pipe
a simple display pipe

struct drm_plane_state *old_state
the plane's old state

Description

During plane updates, this function unpins the GEM VRAM objects of the plane's old framebuffer from VRAM. Complements [drm_gem_vram_simple_display_pipe_prepare_fb\(\)](#).

void **drm_vram_mm_debugfs_init**(struct [drm_minor](#) *minor)
Register VRAM MM debugfs file.

Parameters

struct drm_minor *minor
drm minor device.

int **drmm_vram_helper_init**(struct [drm_device](#) *dev, uint64_t vram_base, size_t vram_size)
Initializes a device's instance of [struct drm_vram_mm](#)

Parameters

struct drm_device *dev
the DRM device

uint64_t vram_base
the base address of the video memory

size_t vram_size
the size of the video memory in bytes

Description

Creates a new instance of [struct drm_vram_mm](#) and stores it in struct [drm_device.vram_mm](#). The instance is auto-managed and cleaned up as part of device cleanup. Calling this function multiple times will generate an error message.

Return

0 on success, or a negative errno code otherwise.

enum [drm_mode_status](#) **drm_vram_helper_mode_valid**(struct [drm_device](#) *dev, const struct [drm_display_mode](#) *mode)

Tests if a display mode's framebuffer fits into the available video memory.

Parameters

struct drm_device *dev
the DRM device

const struct drm_display_mode *mode
the mode to test

Description

This function tests if enough video memory is available for using the specified display mode. Atomic modesetting requires importing the designated framebuffer into video memory before evicting the active one. Hence, any framebuffer may consume at most half of the available VRAM. Display modes that require a larger framebuffer can not be used, even if the CRTC does support them. Each framebuffer is assumed to have 32-bit color depth.

Note

The function can only test if the display mode is supported in general. If there are too many framebuffers pinned to video memory, a display mode may still not be usable in practice. The color depth of 32-bit fits all current use case. A more flexible test can be added when necessary.

Return

MODE_OK if the display mode is supported, or an error code of type *enum drm_mode_status* otherwise.

3.2.12 GEM TTM Helper Functions Reference

This library provides helper functions for gem objects backed by ttm.

```
void drm_gem_ttm_print_info(struct drm_printer *p, unsigned int indent, const struct  
                           drm_gem_object *gem)
```

Print ttm_buffer_object info for debugfs

Parameters

struct drm_printer *p

DRM printer

unsigned int indent

Tab indentation level

const struct drm_gem_object *gem

GEM object

Description

This function can be used as *drm_gem_object_funcs.print_info* callback.

```
int drm_gem_ttm_vmap(struct drm_gem_object *gem, struct iosys_map *map)
```

vmap ttm_buffer_object

Parameters

struct drm_gem_object *gem

GEM object.

struct iosys_map *map

[out] returns the dma-buf mapping.

Description

Maps a GEM object with ttm_bo_vmap(). This function can be used as *drm_gem_object_funcs.vmap* callback.

Return

0 on success, or a negative errno code otherwise.

```
void drm_gem_ttm_vunmap(struct drm_gem_object *gem, struct iosys_map *map)
```

vunmap ttm_buffer_object

Parameters

struct drm_gem_object *gem

GEM object.

struct iosys_map *map
dma-buf mapping.

Description

Unmaps a GEM object with `ttm_bo_vunmap()`. This function can be used as `drm_gem_object_funcs.vmap` callback.

```
int drm_gem_ttm_mmap(struct drm_gem_object *gem, struct vm_area_struct *vma)
    mmap ttm_buffer_object
```

Parameters

struct drm_gem_object *gem
GEM object.

struct vm_area_struct *vma
vm area.

Description

This function can be used as `drm_gem_object_funcs.mmap` callback.

```
int drm_gem_ttm_dumb_map_offset(struct drm_file *file, struct drm_device *dev, uint32_t
    handle, uint64_t *offset)
```

Implements struct `drm_driver.dumb_map_offset`

Parameters

struct drm_file *file
DRM file pointer.

struct drm_device *dev
DRM device.

uint32_t handle
GEM handle

uint64_t *offset
Returns the mapping's memory offset on success

Description

Provides an implementation of struct `drm_driver.dumb_map_offset` for TTM-based GEM drivers. TTM allocates the offset internally and `drm_gem_ttm_dumb_map_offset()` returns it for dumb-buffer implementations.

See struct `drm_driver.dumb_map_offset`.

Return

0 on success, or a negative errno code otherwise.

3.3 VMA Offset Manager

The vma-manager is responsible to map arbitrary driver-dependent memory regions into the linear user address-space. It provides offsets to the caller which can then be used on the `address_space` of the `drm-device`. It takes care to not overlap regions, size them appropriately and to not confuse `mm-core` by inconsistent fake `vm_pgoff` fields. Drivers shouldn't use this for object placement in `VMEM`. This manager should only be used to manage mappings into linear user-space VMs.

We use `drm_mm` as backend to manage object allocations. But it is highly optimized for `alloc/free` calls, not lookups. Hence, we use an `rb-tree` to speed up offset lookups.

You must not use multiple offset managers on a single `address_space`. Otherwise, `mm-core` will be unable to tear down memory mappings as the VM will no longer be linear.

This offset manager works on page-based addresses. That is, every argument and return code (with the exception of `drm_vma_node_offset_addr()`) is given in number of pages, not number of bytes. That means, object sizes and offsets must always be page-aligned (as usual). If you want to get a valid byte-based user-space address for a given offset, please see `drm_vma_node_offset_addr()`.

Additionally to offset management, the vma offset manager also handles access management. For every open-file context that is allowed to access a given node, you must call `drm_vma_node_allow()`. Otherwise, an `mmap()` call on this open-file with the offset of the node will fail with `-EACCES`. To revoke access again, use `drm_vma_node_revoke()`. However, the caller is responsible for destroying already existing mappings, if required.

```
struct drm_vma_offset_node *drm_vma_offset_exact_lookup_locked(struct
                                                                    drm_vma_offset_manager
                                                                    *mgr, unsigned long
                                                                    start, unsigned long
                                                                    pages)
```

Look up node by exact address

Parameters

struct drm_vma_offset_manager *mgr

Manager object

unsigned long start

Start address (page-based, not byte-based)

unsigned long pages

Size of object (page-based)

Description

Same as `drm_vma_offset_lookup_locked()` but does not allow any offset into the node. It only returns the exact object with the given start address.

Return

Node at exact start address **start**.

void drm_vma_offset_lock_lookup(struct drm_vma_offset_manager *mgr)

Lock lookup for extended private use

Parameters

struct drm_vma_offset_manager *mgr

Manager object

Description

Lock VMA manager for extended lookups. Only locked VMA function calls are allowed while holding this lock. All other contexts are blocked from VMA until the lock is released via [drm_vma_offset_unlock_lookup\(\)](#).

Use this if you need to take a reference to the objects returned by [drm_vma_offset_lookup_locked\(\)](#) before releasing this lock again.

This lock must not be used for anything else than extended lookups. You must not call any other VMA helpers while holding this lock.

Note

You're in atomic-context while holding this lock!

void **drm_vma_offset_unlock_lookup**(struct drm_vma_offset_manager *mgr)

Unlock lookup for extended private use

Parameters

struct drm_vma_offset_manager *mgr

Manager object

Description

Release lookup-lock. See [drm_vma_offset_lock_lookup\(\)](#) for more information.

void **drm_vma_node_reset**(struct drm_vma_offset_node *node)

Initialize or reset node object

Parameters

struct drm_vma_offset_node *node

Node to initialize or reset

Description

Reset a node to its initial state. This must be called before using it with any VMA offset manager.

This must not be called on an already allocated node, or you will leak memory.

unsigned long **drm_vma_node_start**(const struct drm_vma_offset_node *node)

Return start address for page-based addressing

Parameters

const struct drm_vma_offset_node *node

Node to inspect

Description

Return the start address of the given node. This can be used as offset into the linear VM space that is provided by the VMA offset manager. Note that this can only be used for page-based addressing. If you need a proper offset for user-space mappings, you must apply "<< PAGE_SHIFT" or use the [drm_vma_node_offset_addr\(\)](#) helper instead.

Return

Start address of **node** for page-based addressing. 0 if the node does not have an offset allocated.

unsigned long **drm_vma_node_size**(struct drm_vma_offset_node *node)

Return size (page-based)

Parameters

struct drm_vma_offset_node *node

Node to inspect

Description

Return the size as number of pages for the given node. This is the same size that was passed to [drm_vma_offset_add\(\)](#). If no offset is allocated for the node, this is 0.

Return

Size of **node** as number of pages. 0 if the node does not have an offset allocated.

__u64 **drm_vma_node_offset_addr**(struct drm_vma_offset_node *node)

Return sanitized offset for user-space mmap

Parameters

struct drm_vma_offset_node *node

Linked offset node

Description

Same as [drm_vma_node_start\(\)](#) but returns the address as a valid offset that can be used for user-space mappings during mmap(). This must not be called on unlinked nodes.

Return

Offset of **node** for byte-based addressing. 0 if the node does not have an object allocated.

void **drm_vma_node_unmap**(struct drm_vma_offset_node *node, struct address_space *file_mapping)

Unmap offset node

Parameters

struct drm_vma_offset_node *node

Offset node

struct address_space *file_mapping

Address space to unmap **node** from

Description

Unmap all userspace mappings for a given offset node. The mappings must be associated with the **file_mapping** address-space. If no offset exists nothing is done.

This call is unlocked. The caller must guarantee that [drm_vma_offset_remove\(\)](#) is not called on this node concurrently.

int **drm_vma_node_verify_access**(struct drm_vma_offset_node *node, struct [drm_file](#) *tag)

Access verification helper for TTM

Parameters

struct drm_vma_offset_node *node

Offset node

struct drm_file *tag

Tag of file to check

Description

This checks whether **tag** is granted access to **node**. It is the same as [drm_vma_node_is_allowed\(\)](#) but suitable as drop-in helper for TTM `verify_access()` call-backs.

Return

0 if access is granted, -EACCES otherwise.

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

Initialize new offset-manager

Parameters

struct drm_vma_offset_manager *mgr

Manager object

unsigned long page_offset

Offset of available memory area (page-based)

unsigned long size

Size of available address space range (page-based)

Description

Initialize a new offset-manager. The offset and area size available for the manager are given as **page_offset** and **size**. Both are interpreted as page-numbers, not bytes.

Adding/removing nodes from the manager is locked internally and protected against concurrent access. However, node allocation and destruction is left for the caller. While calling into the vma-manager, a given node must always be guaranteed to be referenced.

void **drm_vma_offset_manager_destroy**(struct drm_vma_offset_manager *mgr)

Destroy offset manager

Parameters

struct drm_vma_offset_manager *mgr

Manager object

Description

Destroy an object manager which was previously created via [drm_vma_offset_manager_init\(\)](#). The caller must remove all allocated nodes before destroying the manager. Otherwise, `drm_mm` will refuse to free the requested resources.

The manager must not be accessed after this function is called.

struct drm_vma_offset_node ***drm_vma_offset_lookup_locked**(struct
drm_vma_offset_manager
*mgr, unsigned long start,
unsigned long pages)

Find node in offset space

Parameters

struct drm_vma_offset_manager *mgr

Manager object

unsigned long start

Start address for object (page-based)

unsigned long pages

Size of object (page-based)

Description

Find a node given a start address and object size. This returns the `_best_` match for the given node. That is, **start** may point somewhere into a valid region and the given node will be returned, as long as the node spans the whole requested area (given the size in number of pages as **pages**).

Note that before lookup the vma offset manager lookup lock must be acquired with [drm_vma_offset_lock_lookup\(\)](#). See there for an example. This can then be used to implement weakly referenced lookups using `kref_get_unless_zero()`.

```
drm_vma_offset_lock_lookup(mgr);
node = drm_vma_offset_lookup_locked(mgr);
if (node)
    kref_get_unless_zero(container_of(node, sth, entr));
drm_vma_offset_unlock_lookup(mgr);
```

Example

Return

Returns NULL if no suitable node can be found. Otherwise, the best match is returned. It's the caller's responsibility to make sure the node doesn't get destroyed before the caller can access it.

int drm_vma_offset_add(struct drm_vma_offset_manager *mgr, struct drm_vma_offset_node *node, unsigned long pages)

Add offset node to manager

Parameters

struct drm_vma_offset_manager *mgr

Manager object

struct drm_vma_offset_node *node

Node to be added

unsigned long pages

Allocation size visible to user-space (in number of pages)

Description

Add a node to the offset-manager. If the node was already added, this does nothing and return 0. **pages** is the size of the object given in number of pages. After this call succeeds, you can access the offset of the node until it is removed again.

If this call fails, it is safe to retry the operation or call [drm_vma_offset_remove\(\)](#), anyway. However, no cleanup is required in that case.

pages is not required to be the same size as the underlying memory object that you want to map. It only limits the size that user-space can map into their address space.

Return

0 on success, negative error code on failure.

```
void drm_vma_offset_remove(struct drm_vma_offset_manager *mgr, struct  
                           drm_vma_offset_node *node)
```

Remove offset node from manager

Parameters

```
struct drm_vma_offset_manager *mgr  
    Manager object
```

```
struct drm_vma_offset_node *node  
    Node to be removed
```

Description

Remove a node from the offset manager. If the node wasn't added before, this does nothing. After this call returns, the offset and size will be 0 until a new offset is allocated via [drm_vma_offset_add\(\)](#) again. Helper functions like [drm_vma_node_start\(\)](#) and [drm_vma_node_offset_addr\(\)](#) will return 0 if no offset is allocated.

```
int drm_vma_node_allow(struct drm_vma_offset_node *node, struct drm_file *tag)  
    Add open-file to list of allowed users
```

Parameters

```
struct drm_vma_offset_node *node  
    Node to modify
```

```
struct drm_file *tag  
    Tag of file to remove
```

Description

Add **tag** to the list of allowed open-files for this node. If **tag** is already on this list, the ref-count is incremented.

The list of allowed-users is preserved across [drm_vma_offset_add\(\)](#) and [drm_vma_offset_remove\(\)](#) calls. You may even call it if the node is currently not added to any offset-manager.

You must remove all open-files the same number of times as you added them before destroying the node. Otherwise, you will leak memory.

This is locked against concurrent access internally.

Return

0 on success, negative error code on internal failure (out-of-mem)

```
int drm_vma_node_allow_once(struct drm_vma_offset_node *node, struct drm_file *tag)  
    Add open-file to list of allowed users
```

Parameters

```
struct drm_vma_offset_node *node  
    Node to modify
```

struct drm_file *tag

Tag of file to remove

Description

Add **tag** to the list of allowed open-files for this node.

The list of allowed-users is preserved across [drm_vma_offset_add\(\)](#) and [drm_vma_offset_remove\(\)](#) calls. You may even call it if the node is currently not added to any offset-manager.

This is not ref-counted unlike [drm_vma_node_allow\(\)](#) hence [drm_vma_node_revoke\(\)](#) should only be called once after this.

This is locked against concurrent access internally.

Return

0 on success, negative error code on internal failure (out-of-mem)

void **drm_vma_node_revoke**(struct drm_vma_offset_node *node, struct [drm_file](#) *tag)

Remove open-file from list of allowed users

Parameters

struct drm_vma_offset_node *node

Node to modify

struct drm_file *tag

Tag of file to remove

Description

Decrement the ref-count of **tag** in the list of allowed open-files on **node**. If the ref-count drops to zero, remove **tag** from the list. You must call this once for every [drm_vma_node_allow\(\)](#) on **tag**.

This is locked against concurrent access internally.

If **tag** is not on the list, nothing is done.

bool **drm_vma_node_is_allowed**(struct drm_vma_offset_node *node, struct [drm_file](#) *tag)

Check whether an open-file is granted access

Parameters

struct drm_vma_offset_node *node

Node to check

struct drm_file *tag

Tag of file to remove

Description

Search the list in **node** whether **tag** is currently on the list of allowed open-files (see [drm_vma_node_allow\(\)](#)).

This is locked against concurrent access internally.

Return

true if **filp** is on the list

3.4 PRIME Buffer Sharing

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

3.4.1 Overview and Lifetime Rules

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

Drivers that support the PRIME API implement the `drm_gem_object_funcs.export` and `drm_driver.gem_prime_import` hooks. `dma_buf_ops` implementations for drivers are all individually exported for drivers which need to overwrite or reimplement some of them.

Reference Counting for GEM Drivers

On the export the `dma_buf` holds a reference to the exported buffer object, usually a `drm_gem_object`. It takes this reference in the `PRIME_HANDLE_TO_FD` IOCTL, when it first calls `drm_gem_object_funcs.export` and stores the exporting GEM object in the `dma_buf.priv` field. This reference needs to be released when the final reference to the `dma_buf` itself is dropped and its `dma_buf_ops.release` function is called. For GEM-based drivers, the `dma_buf` should be exported using `drm_gem_dmabuf_export()` and then released by `drm_gem_dmabuf_release()`.

Thus the chain of references always flows in one direction, avoiding loops: importing GEM object -> dma-buf -> exported GEM bo. A further complication are the lookup caches for import and export. These are required to guarantee that any given object will always have only one unique userspace handle. This is required to allow userspace to detect duplicated imports, since some GEM drivers do fail command submissions if a given buffer object is listed more than once. These import and export caches in `drm_prime_file_private` only retain a weak reference, which is cleaned up when the corresponding object is released.

Self-importing: If userspace is using PRIME as a replacement for flink then it will get a `fd->handle` request for a GEM object that it created. Drivers should detect this situation and return back the underlying object from the dma-buf private. For GEM based drivers this is handled in `drm_gem_prime_import()` already.

3.4.2 PRIME Helper Functions

Drivers can implement `drm_gem_object_funcs.export` and `drm_driver.gem_prime_import` in terms of simpler APIs by using the helper functions `drm_gem_prime_export()` and `drm_gem_prime_import()`. These functions implement dma-buf support in terms of some lower-level helpers, which are again exported for drivers to use individually:

Exporting buffers

Optional pinning of buffers is handled at dma-buf attach and detach time in `drm_gem_map_attach()` and `drm_gem_map_detach()`. Backing storage itself is handled by `drm_gem_map_dma_buf()` and `drm_gem_unmap_dma_buf()`, which relies on `drm_gem_object_funcs.get_sg_table`. If `drm_gem_object_funcs.get_sg_table` is unimplemented, exports into another device are rejected.

For kernel-internal access there's `drm_gem_dmabuf_vmap()` and `drm_gem_dmabuf_vunmap()`. Userspace mmap support is provided by `drm_gem_dmabuf_mmap()`.

Note that these export helpers can only be used if the underlying backing storage is fully coherent and either permanently pinned, or it is safe to pin it indefinitely.

FIXME: The underlying helper functions are named rather inconsistently.

Importing buffers

Importing dma-bufs using `drm_gem_prime_import()` relies on `drm_driver.gem_prime_import_sg_table`.

Note that similarly to the export helpers this permanently pins the underlying backing storage. Which is ok for scanout, but is not the best option for sharing lots of buffers for rendering.

3.4.3 PRIME Function References

struct **drm_prime_file_private**
per-file tracking for PRIME

Definition:

```
struct drm_prime_file_private {  
};
```

Members

Description

This just contains the internal struct `dma_buf` and handle caches for each `struct drm_file` used by the PRIME core code.

struct `dma_buf` ***drm_gem_dmabuf_export**(struct `drm_device` *dev, struct `dma_buf_export_info` *exp_info)
dma_buf export implementation for GEM

Parameters

struct `drm_device` *dev
parent device for the exported dmabuf

struct `dma_buf_export_info` *exp_info
the export information used by `dma_buf_export()`

Description

This wraps `dma_buf_export()` for use by generic GEM drivers that are using `drm_gem_dmabuf_release()`. In addition to calling `dma_buf_export()`, we take a reference to the `drm_device` and the exported `drm_gem_object` (stored in `dma_buf_export_info.priv`) which is released by `drm_gem_dmabuf_release()`.

Returns the new dmabuf.

```
void drm_gem_dmabuf_release(struct dma_buf *dma_buf)
    dma_buf release implementation for GEM
```

Parameters

```
struct dma_buf *dma_buf
    buffer to be released
```

Description

Generic release function for dma_bufs exported as PRIME buffers. GEM drivers must use this in their `dma_buf_ops` structure as the release callback. `drm_gem_dmabuf_release()` should be used in conjunction with `drm_gem_dmabuf_export()`.

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

PRIME import function for GEM drivers

Parameters

```
struct drm_device *dev
    drm_device to import into
```

```
struct drm_file *file_priv
    drm file-private structure
```

```
int prime_fd
    fd id of the dma-buf which should be imported
```

```
uint32_t *handle
    pointer to storage for the handle of the imported buffer object
```

Description

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

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

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

PRIME export function for GEM drivers

Parameters

```
struct drm_device *dev
    dev to export the buffer from
```

```
struct drm_file *file_priv
    drm file-private structure
```

```
uint32_t handle
    buffer handle to export
```

uint32_t flags

flags like `DRM_CLOEXEC`

int *prime_fd

pointer to storage for the fd id of the create dma-buf

Description

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

int **drm_gem_map_attach**(struct *dma_buf* *dma_buf, struct dma_buf_attachment *attach)
dma_buf attach implementation for GEM

Parameters

struct dma_buf *dma_buf
buffer to attach device to

struct dma_buf_attachment *attach
buffer attachment data

Description

Calls `drm_gem_object_funcs.pin` for device specific handling. This can be used as the `dma_buf_ops.attach` callback. Must be used together with `drm_gem_map_detach()`.

Returns 0 on success, negative error code on failure.

void **drm_gem_map_detach**(struct *dma_buf* *dma_buf, struct dma_buf_attachment *attach)
dma_buf detach implementation for GEM

Parameters

struct dma_buf *dma_buf
buffer to detach from

struct dma_buf_attachment *attach
attachment to be detached

Description

Calls `drm_gem_object_funcs.pin` for device specific handling. Cleans up `dma_buf_attachment` from `drm_gem_map_attach()`. This can be used as the `dma_buf_ops.detach` callback.

struct sg_table ***drm_gem_map_dma_buf**(struct dma_buf_attachment *attach, enum dma_data_direction dir)
map_dma_buf implementation for GEM

Parameters

struct dma_buf_attachment *attach
attachment whose scatterlist is to be returned

enum dma_data_direction dir
direction of DMA transfer

Description

Calls `drm_gem_object_funcs.get_sg_table` and then maps the scatterlist. This can be used as the `dma_buf_ops.map_dma_buf` callback. Should be used together with `drm_gem_unmap_dma_buf()`.

Return

`sg_table` containing the scatterlist to be returned; returns `ERR_PTR` on error. May return `-EINTR` if it is interrupted by a signal.

```
void drm_gem_unmap_dma_buf(struct dma_buf_attachment *attach, struct sg_table *sgt, enum
                           dma_data_direction dir)
    unmap_dma_buf implementation for GEM
```

Parameters

struct dma_buf_attachment *attach
attachment to unmap buffer from

struct sg_table *sgt
scatterlist info of the buffer to unmap

enum dma_data_direction dir
direction of DMA transfer

Description

This can be used as the `dma_buf_ops.unmap_dma_buf` callback.

```
int drm_gem_dmabuf_vmap(struct dma_buf *dma_buf, struct iosys_map *map)
    dma_buf vmap implementation for GEM
```

Parameters

struct dma_buf *dma_buf
buffer to be mapped

struct iosys_map *map
the virtual address of the buffer

Description

Sets up a kernel virtual mapping. This can be used as the `dma_buf_ops.vmap` callback. Calls into `drm_gem_object_funcs.vmap` for device specific handling. The kernel virtual address is returned in `map`.

Returns 0 on success or a negative `errno` code otherwise.

```
void drm_gem_dmabuf_vunmap(struct dma_buf *dma_buf, struct iosys_map *map)
    dma_buf vunmap implementation for GEM
```

Parameters

struct dma_buf *dma_buf
buffer to be unmapped

struct iosys_map *map
the virtual address of the buffer

Description

Releases a kernel virtual mapping. This can be used as the `dma_buf_ops.vunmap` callback. Calls into `drm_gem_object_funcs.vunmap` for device specific handling.

int **drm_gem_prime_mmap**(struct *drm_gem_object* *obj, struct vm_area_struct *vma)
PRIME mmap function for GEM drivers

Parameters

struct drm_gem_object *obj
GEM object

struct vm_area_struct *vma
Virtual address range

Description

This function sets up a userspace mapping for PRIME exported buffers using the same codepath that is used for regular GEM buffer mapping on the DRM fd. The fake GEM offset is added to vma->vm_pgoff and *drm_driver->fops->mmap* is called to set up the mapping.

int **drm_gem_dmabuf_mmap**(struct *dma_buf* *dma_buf, struct vm_area_struct *vma)
dma_buf mmap implementation for GEM

Parameters

struct dma_buf *dma_buf
buffer to be mapped

struct vm_area_struct *vma
virtual address range

Description

Provides memory mapping for the buffer. This can be used as the *dma_buf_ops.mmap* callback. It just forwards to *drm_gem_prime_mmap()*.

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

struct sg_table ***drm_prime_pages_to_sg**(struct *drm_device* *dev, struct page **pages,
unsigned int nr_pages)
converts a page array into an sg list

Parameters

struct drm_device *dev
DRM device

struct page **pages
pointer to the array of page pointers to convert

unsigned int nr_pages
length of the page vector

Description

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

This is useful for implementing *drm_gem_object_funcs.get_sg_table*.

unsigned long **drm_prime_get_contiguous_size**(struct sg_table *sgt)
returns the contiguous size of the buffer

Parameters

struct sg_table *sgt

sg_table describing the buffer to check

Description

This helper calculates the contiguous size in the DMA address space of the buffer described by the provided sg_table.

This is useful for implementing *drm_gem_object_funcs.gem_prime_import_sg_table*.

struct dma_buf *drm_gem_prime_export(struct *drm_gem_object* *obj, int flags)

helper library implementation of the export callback

Parameters

struct drm_gem_object *obj

GEM object to export

int flags

flags like DRM_CLOEXEC and DRM_RDWR

Description

This is the implementation of the *drm_gem_object_funcs.export* functions for GEM drivers using the PRIME helpers. It is used as the default in *drm_gem_prime_handle_to_fd()*.

struct drm_gem_object *drm_gem_prime_import_dev(struct *drm_device* *dev, struct *dma_buf* *dma_buf, struct device *attach_dev)

core implementation of the import callback

Parameters

struct drm_device *dev

drm_device to import into

struct dma_buf *dma_buf

dma-buf object to import

struct device *attach_dev

struct device to dma_buf attach

Description

This is the core of *drm_gem_prime_import()*. It's designed to be called by drivers who want to use a different device structure than *drm_device.dev* for attaching via dma_buf. This function calls *drm_driver.gem_prime_import_sg_table* internally.

Drivers must arrange to call *drm_prime_gem_destroy()* from their *drm_gem_object_funcs.free* hook when using this function.

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

helper library implementation of the import callback

Parameters

struct drm_device *dev

drm_device to import into

struct dma_buf *dma_buf

dma-buf object to import

Description

This is the implementation of the `gem_prime_import` functions for GEM drivers using the PRIME helpers. Drivers can use this as their `drm_driver.gem_prime_import` implementation. It is used as the default implementation in `drm_gem_prime_fd_to_handle()`.

Drivers must arrange to call `drm_prime_gem_destroy()` from their `drm_gem_object_funcs.free` hook when using this function.

int `drm_prime_sg_to_page_array`(struct sg_table *sgt, struct page **pages, int max_entries)
convert an sg table into a page array

Parameters

struct sg_table *sgt
scatter-gather table to convert

struct page **pages
array of page pointers to store the pages in

int max_entries
size of the passed-in array

Description

Exports an sg table into an array of pages.

This function is deprecated and strongly discouraged to be used. The page array is only useful for page faults and those can corrupt fields in the struct page if they are not handled by the exporting driver.

int `drm_prime_sg_to_dma_addr_array`(struct sg_table *sgt, dma_addr_t *addrs, int max_entries)
convert an sg table into a dma addr array

Parameters

struct sg_table *sgt
scatter-gather table to convert

dma_addr_t *addrs
array to store the dma bus address of each page

int max_entries
size of both the passed-in arrays

Description

Exports an sg table into an array of addresses.

Drivers should use this in their `drm_driver.gem_prime_import_sg_table` implementation.

void `drm_prime_gem_destroy`(struct *drm_gem_object* *obj, struct sg_table *sg)
helper to clean up a PRIME-imported GEM object

Parameters

struct *drm_gem_object* *obj
GEM object which was created from a dma-buf

struct sg_table *sg
the sg-table which was pinned at import time

Description

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

3.5 DRM MM Range Allocator

3.5.1 Overview

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

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

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

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

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

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

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

Note that this range allocator is not thread-safe, drivers need to protect modifications with their own locking. The idea behind this is that for a full memory manager additional data needs to be protected anyway, hence internal locking would be fully redundant.

3.5.2 LRU Scan/Eviction Support

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

The DRM range allocator supports this use-case through the scanning interfaces. First a scan operation needs to be initialized with `drm_mm_scan_init()` or `drm_mm_scan_init_with_range()`. The driver adds objects to the roster, probably by walking an LRU list, but this can be freely implemented. Eviction candidates are added using `drm_mm_scan_add_block()` until a suitable hole is found or there are no further evictable objects. Eviction roster metadata is tracked in `struct drm_mm_scan`.

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

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

3.5.3 DRM MM Range Allocator Function References

enum **drm_mm_insert_mode**

control search and allocation behaviour

Constants

DRM_MM_INSERT_BEST

Search for the smallest hole (within the search range) that fits the desired node.

Allocates the node from the bottom of the found hole.

DRM_MM_INSERT_LOW

Search for the lowest hole (address closest to 0, within the search range) that fits the desired node.

Allocates the node from the bottom of the found hole.

DRM_MM_INSERT_HIGH

Search for the highest hole (address closest to `U64_MAX`, within the search range) that fits the desired node.

Allocates the node from the *top* of the found hole. The specified alignment for the node is applied to the base of the node (`drm_mm_node.start`).

DRM_MM_INSERT_EVICT

Search for the most recently evicted hole (within the search range) that fits the desired node. This is appropriate for use immediately after performing an eviction scan (see `drm_mm_scan_init()`) and removing the selected nodes to form a hole.

Allocates the node from the bottom of the found hole.

DRM_MM_INSERT_ONCE

Only check the first hole for suitability and report -ENOSPC immediately otherwise, rather than check every hole until a suitable one is found. Can only be used in conjunction with another search method such as `DRM_MM_INSERT_HIGH` or `DRM_MM_INSERT_LOW`.

DRM_MM_INSERT_HIGHEST

Only check the highest hole (the hole with the largest address) and insert the node at the top of the hole or report -ENOSPC if unsuitable.

Does not search all holes.

DRM_MM_INSERT_LOWEST

Only check the lowest hole (the hole with the smallest address) and insert the node at the bottom of the hole or report -ENOSPC if unsuitable.

Does not search all holes.

Description

The `struct drm_mm` range manager supports finding a suitable modes using a number of search trees. These trees are organised by size, by address and in most recent eviction order. This allows the user to find either the smallest hole to reuse, the lowest or highest address to reuse, or simply reuse the most recent eviction that fits. When allocating the `drm_mm_node` from within the hole, the `drm_mm_insert_mode` also dictate whether to allocate the lowest matching address or the highest.

struct **drm_mm_node**

allocated block in the DRM allocator

Definition:

```
struct drm_mm_node {
    unsigned long color;
    u64 start;
    u64 size;
};
```

Members**color**

Opaque driver-private tag.

start

Start address of the allocated block.

size

Size of the allocated block.

Description

This represents an allocated block in a `drm_mm` allocator. Except for pre-reserved nodes inserted using `drm_mm_reserve_node()` the structure is entirely opaque and should only be accessed through the provided functions. Since allocation of these nodes is entirely handled by the driver they can be embedded.

struct **drm_mm**

DRM allocator

Definition:

```
struct drm_mm {
    void (*color_adjust)(const struct drm_mm_node *node,unsigned long color,
↪u64 *start, u64 *end);
};
```

Members

color_adjust

Optional driver callback to further apply restrictions on a hole. The node argument points at the node containing the hole from which the block would be allocated (see [drm_mm_hole_follows\(\)](#) and friends). The other arguments are the size of the block to be allocated. The driver can adjust the start and end as needed to e.g. insert guard pages.

Description

DRM range allocator with a few special functions and features geared towards managing GPU memory. Except for the **color_adjust** callback the structure is entirely opaque and should only be accessed through the provided functions and macros. This structure can be embedded into larger driver structures.

struct **drm_mm_scan**

DRM allocator eviction roaster data

Definition:

```
struct drm_mm_scan {
};
```

Members

Description

This structure tracks data needed for the eviction roaster set up using [drm_mm_scan_init\(\)](#), and used with [drm_mm_scan_add_block\(\)](#) and [drm_mm_scan_remove_block\(\)](#). The structure is entirely opaque and should only be accessed through the provided functions and macros. It is meant to be allocated temporarily by the driver on the stack.

bool **drm_mm_node_allocated**(const struct [drm_mm_node](#) *node)

checks whether a node is allocated

Parameters

const struct **drm_mm_node** *node

drm_mm_node to check

Description

Drivers are required to clear a node prior to using it with the `drm_mm` range manager.

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

Return

True if the **node** is allocated.

bool **drm_mm_initialized**(const struct [drm_mm](#) *mm)

checks whether an allocator is initialized

Parameters

const struct drm_mm *mm
drm_mm to check

Description

Drivers should clear the *struct drm_mm* prior to initialisation if they want to use this function. Drivers should use this helper for proper encapsulation of *drm_mm* internals.

Return

True if the **mm** is initialized.

bool **drm_mm_hole_follows**(const struct *drm_mm_node* *node)
checks whether a hole follows this node

Parameters

const struct drm_mm_node *node
drm_mm_node to check

Description

Holes are embedded into the *drm_mm* using the tail of a *drm_mm_node*. If you wish to know whether a hole follows this particular node, query this function. See also *drm_mm_hole_node_start()* and *drm_mm_hole_node_end()*.

Return

True if a hole follows the **node**.

u64 **drm_mm_hole_node_start**(const struct *drm_mm_node* *hole_node)
computes the start of the hole following **node**

Parameters

const struct drm_mm_node *hole_node
drm_mm_node which implicitly tracks the following hole

Description

This is useful for driver-specific debug dumpers. Otherwise drivers should not inspect holes themselves. Drivers must check first whether a hole indeed follows by looking at *drm_mm_hole_follows()*

Return

Start of the subsequent hole.

u64 **drm_mm_hole_node_end**(const struct *drm_mm_node* *hole_node)
computes the end of the hole following **node**

Parameters

const struct drm_mm_node *hole_node
drm_mm_node which implicitly tracks the following hole

Description

This is useful for driver-specific debug dumpers. Otherwise drivers should not inspect holes themselves. Drivers must check first whether a hole indeed follows by looking at *drm_mm_hole_follows()*.

Return

End of the subsequent hole.

drm_mm_nodes

drm_mm_nodes (mm)

list of nodes under the drm_mm range manager

Parameters

mm

the *struct drm_mm* range manager

Description

As the drm_mm range manager hides its node_list deep with its structure, extracting it looks painful and repetitive. This is not expected to be used outside of the *drm_mm_for_each_node()* macros and similar internal functions.

Return

The node list, may be empty.

drm_mm_for_each_node

drm_mm_for_each_node (entry, mm)

iterator to walk over all allocated nodes

Parameters

entry

struct drm_mm_node to assign to in each iteration step

mm

drm_mm allocator to walk

Description

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

drm_mm_for_each_node_safe

drm_mm_for_each_node_safe (entry, next, mm)

iterator to walk over all allocated nodes

Parameters

entry

struct drm_mm_node to assign to in each iteration step

next

struct drm_mm_node to store the next step

mm

drm_mm allocator to walk

Description

This iterator walks over all nodes in the range allocator. It is implemented with `list_for_each_safe()`, so save against removal of elements.

drm_mm_for_each_hole

`drm_mm_for_each_hole (pos, mm, hole_start, hole_end)`
iterator to walk over all holes

Parameters

pos

drm_mm_node used internally to track progress

mm

drm_mm allocator to walk

hole_start

ulong variable to assign the hole start to on each iteration

hole_end

ulong variable to assign the hole end to on each iteration

Description

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

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

```
int drm_mm_insert_node_generic(struct drm_mm *mm, struct drm_mm_node *node, u64
                                size, u64 alignment, unsigned long color, enum
                                drm_mm_insert_mode mode)
```

search for space and insert **node**

Parameters

struct *drm_mm* *mm

drm_mm to allocate from

struct *drm_mm_node* *node

preallocate node to insert

u64 size

size of the allocation

u64 alignment

alignment of the allocation

unsigned long color

opaque tag value to use for this node

enum *drm_mm_insert_mode* mode

fine-tune the allocation search and placement

Description

This is a simplified version of `drm_mm_insert_node_in_range()` with no range restrictions applied.

The preallocated node must be cleared to 0.

Return

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

int **drm_mm_insert_node**(struct *drm_mm* *mm, struct *drm_mm_node* *node, u64 size)
search for space and insert **node**

Parameters

struct drm_mm *mm
drm_mm to allocate from

struct drm_mm_node *node
preallocate node to insert

u64 size
size of the allocation

Description

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

The preallocated node must be cleared to 0.

Return

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

bool **drm_mm_clean**(const struct *drm_mm* *mm)
checks whether an allocator is clean

Parameters

const struct drm_mm *mm
drm_mm allocator to check

Return

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

drm_mm_for_each_node_in_range

drm_mm_for_each_node_in_range (node__, mm__, start__, end__)
iterator to walk over a range of allocated nodes

Parameters

node__
drm_mm_node structure to assign to in each iteration step

mm__
drm_mm allocator to walk

start__
starting offset, the first node will overlap this

end__
ending offset, the last node will start before this (but may overlap)

Description

This iterator walks over all nodes in the range allocator that lie between **start** and **end**. It is implemented similarly to `list_for_each()`, but using the internal interval tree to accelerate the search for the starting node, and so not safe against removal of elements. It assumes that **end** is within (or is the upper limit of) the `drm_mm` allocator. If [**start**, **end**] are beyond the range of the `drm_mm`, the iterator may walk over the special `_unallocated_` `drm_mm.head_node`, and may even continue indefinitely.

```
void drm_mm_scan_init(struct drm_mm_scan *scan, struct drm_mm *mm, u64 size, u64
                      alignment, unsigned long color, enum drm_mm_insert_mode mode)
    initialize lru scanning
```

Parameters

```
struct drm_mm_scan *scan
    scan state

struct drm_mm *mm
    drm_mm to scan

u64 size
    size of the allocation

u64 alignment
    alignment of the allocation

unsigned long color
    opaque tag value to use for the allocation

enum drm_mm_insert_mode mode
    fine-tune the allocation search and placement
```

Description

This is a simplified version of `drm_mm_scan_init_with_range()` with no range restrictions applied.

This simply sets up the scanning routines with the parameters for the desired hole.

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

```
int drm_mm_reserve_node(struct drm_mm *mm, struct drm_mm_node *node)
    insert an pre-initialized node
```

Parameters

```
struct drm_mm *mm
    drm_mm allocator to insert node into

struct drm_mm_node *node
    drm_mm_node to insert
```

Description

This functions inserts an already set-up `drm_mm_node` into the allocator, meaning that start, size and color must be set by the caller. All other fields must be cleared to 0. This is useful to initialize the allocator with preallocated objects which must be set-up before the range allocator can be set-up, e.g. when taking over a firmware framebuffer.

Return

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

int **drm_mm_insert_node_in_range**(struct *drm_mm* *const mm, struct *drm_mm_node* *const node, u64 size, u64 alignment, unsigned long color, u64 range_start, u64 range_end, enum *drm_mm_insert_mode* mode)

 ranged search for space and insert **node**

Parameters

struct drm_mm * const mm
 drm_mm to allocate from

struct drm_mm_node * const node
 preallocate node to insert

u64 size
 size of the allocation

u64 alignment
 alignment of the allocation

unsigned long color
 opaque tag value to use for this node

u64 range_start
 start of the allowed range for this node

u64 range_end
 end of the allowed range for this node

enum drm_mm_insert_mode mode
 fine-tune the allocation search and placement

Description

The preallocated **node** must be cleared to 0.

Return

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

void **drm_mm_remove_node**(struct *drm_mm_node* *node)
 Remove a memory node from the allocator.

Parameters

struct drm_mm_node *node
 drm_mm_node to remove

Description

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

void **drm_mm_replace_node**(struct *drm_mm_node* *old, struct *drm_mm_node* *new)
 move an allocation from **old** to **new**

Parameters

struct drm_mm_node *old

drm_mm_node to remove from the allocator

struct drm_mm_node *new

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

Description

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

void **drm_mm_scan_init_with_range**(struct *drm_mm_scan* *scan, struct *drm_mm* *mm, u64 size, u64 alignment, unsigned long color, u64 start, u64 end, enum *drm_mm_insert_mode* mode)

initialize range-restricted lru scanning

Parameters

struct drm_mm_scan *scan

scan state

struct drm_mm *mm

drm_mm to scan

u64 size

size of the allocation

u64 alignment

alignment of the allocation

unsigned long color

opaque tag value to use for the allocation

u64 start

start of the allowed range for the allocation

u64 end

end of the allowed range for the allocation

enum drm_mm_insert_mode mode

fine-tune the allocation search and placement

Description

This simply sets up the scanning routines with the parameters for the desired hole.

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

bool **drm_mm_scan_add_block**(struct *drm_mm_scan* *scan, struct *drm_mm_node* *node)

add a node to the scan list

Parameters

struct drm_mm_scan *scan

the active `drm_mm` scanner

struct drm_mm_node *node

drm_mm_node to add

Description

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

Return

True if a hole has been found, false otherwise.

bool **drm_mm_scan_remove_block**(struct *drm_mm_scan* *scan, struct *drm_mm_node* *node)
remove a node from the scan list

Parameters

struct **drm_mm_scan** *scan
the active drm_mm scanner

struct **drm_mm_node** *node
drm_mm_node to remove

Description

Nodes **must** be removed in exactly the reverse order from the scan list as they have been added (e.g. using `list_add()` as they are added and then `list_for_each()` over that eviction list to remove), otherwise the internal state of the memory manager will be corrupted.

When the scan list is empty, the selected memory nodes can be freed. An immediately following `drm_mm_insert_node_in_range_generic()` or one of the simpler versions of that function with `!DRM_MM_SEARCH_BEST` will then return the just freed block (because it's at the top of the `free_stack` list).

Return

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

struct *drm_mm_node* ***drm_mm_scan_color_evict**(struct *drm_mm_scan* *scan)
evict overlapping nodes on either side of hole

Parameters

struct **drm_mm_scan** *scan
drm_mm scan with target hole

Description

After completing an eviction scan and removing the selected nodes, we may need to remove a few more nodes from either side of the target hole if `mm.color_adjust` is being used.

Return

A node to evict, or NULL if there are no overlapping nodes.

void **drm_mm_init**(struct *drm_mm* *mm, u64 start, u64 size)
initialize a drm-mm allocator

Parameters

struct **drm_mm** *mm
the drm_mm structure to initialize

u64 **start**
start of the range managed by **mm**

u64 sizeend of the range managed by **mm****Description**

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

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

clean up a **drm_mm** allocator**Parameters****struct drm_mm *mm****drm_mm** allocator to clean up**Description**

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

```
void drm_mm_print(const struct drm_mm *mm, struct drm_printer *p)
```

print allocator state

Parameters**const struct drm_mm *mm****drm_mm** allocator to print**struct drm_printer *p**

DRM printer to use

3.6 DRM GPU VA Manager

3.6.1 Overview

The DRM GPU VA Manager, represented by *struct drm_gpuva_manager* keeps track of a GPU's virtual address (VA) space and manages the corresponding virtual mappings represented by *drm_gpuva* objects. It also keeps track of the mapping's backing *drm_gem_object* buffers.

drm_gem_object buffers maintain a list of *drm_gpuva* objects representing all existent GPU VA mappings using this *drm_gem_object* as backing buffer.

GPU VAs can be flagged as sparse, such that drivers may use GPU VAs to also keep track of sparse PTEs in order to support Vulkan 'Sparse Resources'.

The GPU VA manager internally uses a rb-tree to manage the *drm_gpuva* mappings within a GPU's virtual address space.

The *drm_gpuva_manager* contains a special *drm_gpuva* representing the portion of VA space reserved by the kernel. This node is initialized together with the GPU VA manager instance and removed when the GPU VA manager is destroyed.

In a typical application drivers would embed *struct drm_gpuva_manager* and *struct drm_gpuva* within their own driver specific structures, there won't be any memory allocations of its own nor memory allocations of *drm_gpuva* entries.

The data structures needed to store *drm_gpuvas* within the *drm_gpuva_manager* are contained within *struct drm_gpuva* already. Hence, for inserting *drm_gpuva* entries from within dma-fence signalling critical sections it is enough to pre-allocate the *drm_gpuva* structures.

3.6.2 Split and Merge

Besides its capability to manage and represent a GPU VA space, the `drm_gpuva_manager` also provides functions to let the `drm_gpuva_manager` calculate a sequence of operations to satisfy a given map or unmap request.

Therefore the DRM GPU VA manager provides an algorithm implementing splitting and merging of existent GPU VA mappings with the ones that are requested to be mapped or unmapped. This feature is required by the Vulkan API to implement Vulkan ‘Sparse Memory Bindings’ - drivers UAPIs often refer to this as VM BIND.

Drivers can call `drm_gpuva_sm_map()` to receive a sequence of callbacks containing map, unmap and remap operations for a given newly requested mapping. The sequence of callbacks represents the set of operations to execute in order to integrate the new mapping cleanly into the current state of the GPU VA space.

Depending on how the new GPU VA mapping intersects with the existent mappings of the GPU VA space the `drm_gpuva_fn_ops` callbacks contain an arbitrary amount of unmap operations, a maximum of two remap operations and a single map operation. The caller might receive no callback at all if no operation is required, e.g. if the requested mapping already exists in the exact same way.

The single map operation represents the original map operation requested by the caller.

`drm_gpuva_op_unmap` contains a ‘keep’ field, which indicates whether the `drm_gpuva` to unmap is physically contiguous with the original mapping request. Optionally, if ‘keep’ is set, drivers may keep the actual page table entries for this `drm_gpuva`, adding the missing page table entries only and update the `drm_gpuva_manager`’s view of things accordingly.

Drivers may do the same optimization, namely delta page table updates, also for remap operations. This is possible since `drm_gpuva_op_remap` consists of one unmap operation and one or two map operations, such that drivers can derive the page table update delta accordingly.

Note that there can’t be more than two existent mappings to split up, one at the beginning and one at the end of the new mapping, hence there is a maximum of two remap operations.

Analogous to `drm_gpuva_sm_map()` `drm_gpuva_sm_unmap()` uses `drm_gpuva_fn_ops` to call back into the driver in order to unmap a range of GPU VA space. The logic behind this function is way simpler though: For all existent mappings enclosed by the given range unmap operations are created. For mappings which are only partially located within the given range, remap operations are created such that those mappings are split up and re-mapped partially.

As an alternative to `drm_gpuva_sm_map()` and `drm_gpuva_sm_unmap()`, `drm_gpuva_sm_map_ops_create()` and `drm_gpuva_sm_unmap_ops_create()` can be used to directly obtain an instance of `struct drm_gpuva_ops` containing a list of `drm_gpuva_op`, which can be iterated with `drm_gpuva_for_each_op()`. This list contains the `drm_gpuva_ops` analogous to the callbacks one would receive when calling `drm_gpuva_sm_map()` or `drm_gpuva_sm_unmap()`. While this way requires more memory (to allocate the `drm_gpuva_ops`), it provides drivers a way to iterate the `drm_gpuva_op` multiple times, e.g. once in a context where memory allocations are possible (e.g. to allocate GPU page tables) and once in the dma-fence signalling critical path.

To update the `drm_gpuva_manager`’s view of the GPU VA space `drm_gpuva_insert()` and `drm_gpuva_remove()` may be used. These functions can safely be used from `drm_gpuva_fn_ops` callbacks originating from `drm_gpuva_sm_map()` or `drm_gpuva_sm_unmap()`. However,

it might be more convenient to use the provided helper functions `drm_gpuva_map()`, `drm_gpuva_remap()` and `drm_gpuva_unmap()` instead.

The following diagram depicts the basic relationships of existent GPU VA mappings, a newly requested mapping and the resulting mappings as implemented by `drm_gpuva_sm_map()` - it doesn't cover any arbitrary combinations of these.

- 1) Requested mapping is identical. Replace it, but indicate the backing PTEs could be kept.

```

      0      a      1
old:  |-----| (bo_offset=n)

      0      a      1
req:  |-----| (bo_offset=n)

      0      a      1
new:  |-----| (bo_offset=n)
```

- 2) Requested mapping is identical, except for the BO offset, hence replace the mapping.

```

      0      a      1
old:  |-----| (bo_offset=n)

      0      a      1
req:  |-----| (bo_offset=m)

      0      a      1
new:  |-----| (bo_offset=m)
```

- 3) Requested mapping is identical, except for the backing BO, hence replace the mapping.

```

      0      a      1
old:  |-----| (bo_offset=n)

      0      b      1
req:  |-----| (bo_offset=n)

      0      b      1
new:  |-----| (bo_offset=n)
```

- 4) Existent mapping is a left aligned subset of the requested one, hence replace the existent one.

```

      0  a  1
old:  |-----| (bo_offset=n)

      0      a      2
req:  |-----| (bo_offset=n)

      0      a      2
new:  |-----| (bo_offset=n)
```

Note: We expect to see the same result for a request with a different BO and/or non-contiguous BO offset.

- 5) Requested mapping's range is a left aligned subset of the existent one, but backed by a different BO. Hence, map the requested mapping and split the existent one adjusting its BO offset.

```

      0      a      2
old: |-----| (bo_offset=n)

      0  b  1
req: |-----| (bo_offset=n)

      0  b  1  a'  2
new: |-----|-----| (b.bo_offset=n, a.bo_offset=n+1)
```

Note: We expect to see the same result for a request with a different BO and/or non-contiguous BO offset.

- 6) Existent mapping is a superset of the requested mapping. Split it up, but indicate that the backing PTEs could be kept.

```

      0      a      2
old: |-----| (bo_offset=n)

      0  a  1
req: |-----| (bo_offset=n)

      0  a  1  a'  2
new: |-----|-----| (a.bo_offset=n, a'.bo_offset=n+1)
```

- 7) Requested mapping's range is a right aligned subset of the existent one, but backed by a different BO. Hence, map the requested mapping and split the existent one, without adjusting the BO offset.

```

      0      a      2
old: |-----| (bo_offset=n)

           1  b  2
req:      |-----| (bo_offset=m)

      0  a  1  b  2
new: |-----|-----| (a.bo_offset=n, b.bo_offset=m)
```

- 8) Existent mapping is a superset of the requested mapping. Split it up, but indicate that the backing PTEs could be kept.

```

      0      a      2
old: |-----| (bo_offset=n)
```



```

      1  a  2
req:  |-----| (bo_offset=n+1)

      0  a' 1  a  2
new:  |-----|-----| (a'.bo_offset=n, a.bo_offset=n+1)

```

- 9) Existent mapping is overlapped at the end by the requested mapping backed by a different BO. Hence, map the requested mapping and split up the existent one, without adjusting the BO offset.

```

      0      a      2
old:  |-----| (bo_offset=n)

      1      b      3
req:  |-----| (bo_offset=m)

      0  a  1      b      3
new:  |-----|-----| (a.bo_offset=n, b.bo_offset=m)

```

- 10) Existent mapping is overlapped by the requested mapping, both having the same backing BO with a contiguous offset. Indicate the backing PTEs of the old mapping could be kept.

```

      0      a      2
old:  |-----| (bo_offset=n)

      1      a      3
req:  |-----| (bo_offset=n+1)

      0  a' 1      a      3
new:  |-----|-----| (a'.bo_offset=n, a.bo_offset=n+1)

```

- 11) Requested mapping's range is a centered subset of the existent one having a different backing BO. Hence, map the requested mapping and split up the existent one in two mappings, adjusting the BO offset of the right one accordingly.

```

      0      a      3
old:  |-----| (bo_offset=n)

      1  b  2
req:  |-----| (bo_offset=m)

      0  a  1  b  2  a' 3
new:  |-----|-----| (a.bo_offset=n, b.bo_offset=m, a'.bo_offset=n+2)

```

- 12) Requested mapping is a contiguous subset of the existent one. Split it up, but indicate that the backing PTEs could be kept.

```

      0      a      3
old:  |-----| (bo_offset=n)

      1  a  2

```

```

req:      |-----|      (bo_offset=n+1)

      0  a' 1  a  2  a'' 3
old: |-----|-----|-----| (a'.bo_offset=n, a.bo_offset=n+1, a''.bo_
      ↪offset=n+2)

```

- 13) Existent mapping is a right aligned subset of the requested one, hence replace the existent one.

```

      1  a  2
old:  |-----| (bo_offset=n+1)

      0      a      2
req:  |-----| (bo_offset=n)

      0      a      2
new:  |-----| (bo_offset=n)

```

Note: We expect to see the same result for a request with a different bo and/or non-contiguous bo_offset.

- 14) Existent mapping is a centered subset of the requested one, hence replace the existent one.

```

      1  a  2
old:  |-----| (bo_offset=n+1)

      0      a      3
req:  |-----| (bo_offset=n)

      0      a      3
new:  |-----| (bo_offset=n)

```

Note: We expect to see the same result for a request with a different bo and/or non-contiguous bo_offset.

- 15) Existent mappings is overlapped at the beginning by the requested mapping backed by a different BO. Hence, map the requested mapping and split up the existent one, adjusting its BO offset accordingly.

```

      1      a      3
old:  |-----| (bo_offset=n)

      0      b      2
req:  |-----| (bo_offset=m)

      0      b      2  a' 3
new:  |-----|-----| (b.bo_offset=m, a.bo_offset=n+2)

```

3.6.3 Locking

Generally, the GPU VA manager does not take care of locking itself, it is the drivers responsibility to take care about locking. Drivers might want to protect the following operations: inserting, removing and iterating *drm_gpuva* objects as well as generating all kinds of operations, such as split / merge or prefetch.

The GPU VA manager also does not take care of the locking of the backing *drm_gem_object* buffers GPU VA lists by itself; drivers are responsible to enforce mutual exclusion using either the GEMs *dma_resv* lock or alternatively a driver specific external lock. For the latter see also *drm_gem_gpuva_set_lock()*.

However, the GPU VA manager contains lockdep checks to ensure callers of its API hold the corresponding lock whenever the *drm_gem_objects* GPU VA list is accessed by functions such as *drm_gpuva_link()* or *drm_gpuva_unlink()*.

3.6.4 Examples

This section gives two examples on how to let the DRM GPUVA Manager generate *drm_gpuva_op* in order to satisfy a given map or unmap request and how to make use of them.

The below code is strictly limited to illustrate the generic usage pattern. To maintain simplicity, it doesn't make use of any abstractions for common code, different (asynchronous) stages with fence signalling critical paths, any other helpers or error handling in terms of freeing memory and dropping previously taken locks.

- 1) Obtain a list of *drm_gpuva_op* to create a new mapping:

```
// Allocates a new &drm_gpuva.
struct drm_gpuva * driver_gpuva_alloc(void);

// Typically drivers would embedd the &drm_gpuva_manager and &drm_gpuva
// structure in individual driver structures and lock the dma-resv with
// drm_exec or similar helpers.
int driver_mapping_create(struct drm_gpuva_manager *mgr,
                        u64 addr, u64 range,
                        struct drm_gem_object *obj, u64 offset)
{
    struct drm_gpuva_ops *ops;
    struct drm_gpuva_op *op

    driver_lock_va_space();
    ops = drm_gpuva_sm_map_ops_create(mgr, addr, range,
                                     obj, offset);

    if (IS_ERR(ops))
        return PTR_ERR(ops);

    drm_gpuva_for_each_op(op, ops) {
        struct drm_gpuva *va;

        switch (op->op) {
        case DRM_GPUVA_OP_MAP:
            va = driver_gpuva_alloc();
```

```
        if (!va)
            ; // unwind previous VA space updates,
            // free memory and unlock

        driver_vm_map();
        drm_gpuva_map(mgr, va, &op->map);
        drm_gpuva_link(va);

        break;
case DRM_GPUVA_OP_REMAP: {
    struct drm_gpuva *prev = NULL, *next = NULL;

    va = op->remap.unmap->va;

    if (op->remap.prev) {
        prev = driver_gpuva_alloc();
        if (!prev)
            ; // unwind previous VA space
            // updates, free memory and
            // unlock
    }

    if (op->remap.next) {
        next = driver_gpuva_alloc();
        if (!next)
            ; // unwind previous VA space
            // updates, free memory and
            // unlock
    }

    driver_vm_remap();
    drm_gpuva_remap(prev, next, &op->remap);

    drm_gpuva_unlink(va);
    if (prev)
        drm_gpuva_link(prev);
    if (next)
        drm_gpuva_link(next);

    break;
}
case DRM_GPUVA_OP_UNMAP:
    va = op->unmap->va;

    driver_vm_unmap();
    drm_gpuva_unlink(va);
    drm_gpuva_unmap(&op->unmap);

    break;
default:
```

```

        break;
    }
}
driver_unlock_va_space();

return 0;
}

```

2) Receive a callback for each `drm_gpuva_op` to create a new mapping:

```

struct driver_context {
    struct drm_gpuva_manager *mgr;
    struct drm_gpuva *new_va;
    struct drm_gpuva *prev_va;
    struct drm_gpuva *next_va;
};

// ops to pass to drm_gpuva_manager_init()
static const struct drm_gpuva_fn_ops driver_gpuva_ops = {
    .sm_step_map = driver_gpuva_map,
    .sm_step_remap = driver_gpuva_remap,
    .sm_step_unmap = driver_gpuva_unmap,
};

// Typically drivers would embedd the &drm_gpuva_manager and &drm_gpuva
// structure in individual driver structures and lock the dma-resv with
// drm_exec or similar helpers.
int driver_mapping_create(struct drm_gpuva_manager *mgr,
                        u64 addr, u64 range,
                        struct drm_gem_object *obj, u64 offset)
{
    struct driver_context ctx;
    struct drm_gpuva_ops *ops;
    struct drm_gpuva_op *op;
    int ret = 0;

    ctx.mgr = mgr;

    ctx.new_va = kzalloc(sizeof(*ctx.new_va), GFP_KERNEL);
    ctx.prev_va = kzalloc(sizeof(*ctx.prev_va), GFP_KERNEL);
    ctx.next_va = kzalloc(sizeof(*ctx.next_va), GFP_KERNEL);
    if (!ctx.new_va || !ctx.prev_va || !ctx.next_va) {
        ret = -ENOMEM;
        goto out;
    }

    driver_lock_va_space();
    ret = drm_gpuva_sm_map(mgr, &ctx, addr, range, obj, offset);
    driver_unlock_va_space();

out:

```

```
        kfree(ctx.new_va);
        kfree(ctx.prev_va);
        kfree(ctx.next_va);
        return ret;
}

int driver_gpuva_map(struct drm_gpuva_op *op, void *__ctx)
{
    struct driver_context *ctx = __ctx;

    drm_gpuva_map(ctx->mgr, ctx->new_va, &op->map);

    drm_gpuva_link(ctx->new_va);

    // prevent the new GPUVA from being freed in
    // driver_mapping_create()
    ctx->new_va = NULL;

    return 0;
}

int driver_gpuva_remap(struct drm_gpuva_op *op, void *__ctx)
{
    struct driver_context *ctx = __ctx;

    drm_gpuva_remap(ctx->prev_va, ctx->next_va, &op->remap);

    drm_gpuva_unlink(op->remap.unmap->va);
    kfree(op->remap.unmap->va);

    if (op->remap.prev) {
        drm_gpuva_link(ctx->prev_va);
        ctx->prev_va = NULL;
    }

    if (op->remap.next) {
        drm_gpuva_link(ctx->next_va);
        ctx->next_va = NULL;
    }

    return 0;
}

int driver_gpuva_unmap(struct drm_gpuva_op *op, void *__ctx)
{
    drm_gpuva_unlink(op->unmap.va);
    drm_gpuva_unmap(&op->unmap);
    kfree(op->unmap.va);

    return 0;
}
```

```
}
```

3.6.5 DRM GPU VA Manager Function References

enum **drm_gpuva_flags**

flags for *struct drm_gpuva*

Constants

DRM_GPUVA_INVALIDATED

Flag indicating that the *drm_gpuva*'s backing GEM is invalidated.

DRM_GPUVA_SPARSE

Flag indicating that the *drm_gpuva* is a sparse mapping.

DRM_GPUVA_USERBITS

user defined bits

struct **drm_gpuva**

structure to track a GPU VA mapping

Definition:

```
struct drm_gpuva {
    struct drm_gpuva_manager *mgr;
    enum drm_gpuva_flags flags;
    struct {
        u64 addr;
        u64 range;
    } va;
    struct {
        u64 offset;
        struct drm_gem_object *obj;
        struct list_head entry;
    } gem;
    struct {
        struct rb_node node;
        struct list_head entry;
        u64 __subtree_last;
    } rb;
};
```

Members

mgr

the *drm_gpuva_manager* this object is associated with

flags

the *drm_gpuva_flags* for this mapping

va

structure containing the address and range of the *drm_gpuva*

gem

structure containing the *drm_gem_object* and it's offset

rb

the rb-tree node

Description

This structure represents a GPU VA mapping and is associated with a *drm_gpuva_manager*. Typically, this structure is embedded in bigger driver structures.

void **drm_gpuva_invalidate**(struct *drm_gpuva* *va, bool invalidate)
sets whether the backing GEM of this *drm_gpuva* is invalidated

Parameters

struct *drm_gpuva* *va
the *drm_gpuva* to set the invalidate flag for

bool **invalidate**
indicates whether the *drm_gpuva* is invalidated

bool **drm_gpuva_invalidated**(struct *drm_gpuva* *va)
indicates whether the backing BO of this *drm_gpuva* is invalidated

Parameters

struct *drm_gpuva* *va
the *drm_gpuva* to check

struct *drm_gpuva_manager*
DRM GPU VA Manager

Definition:

```
struct drm_gpuva_manager {
    const char *name;
    u64 mm_start;
    u64 mm_range;
    struct {
        struct rb_root_cached tree;
        struct list_head list;
    } rb;
    struct drm_gpuva kernel_alloc_node;
    const struct drm_gpuva_fn_ops *ops;
};
```

Members

name
the name of the DRM GPU VA space

mm_start
start of the VA space

mm_range
length of the VA space

rbstructures to track *drm_gpuva* entries**kernel_alloc_node***drm_gpuva* representing the address space cutout reserved for the kernel**ops***drm_gpuva_fn_ops* providing the split/merge steps to drivers**Description**

The DRM GPU VA Manager keeps track of a GPU's virtual address space by using *maple_tree* structures. Typically, this structure is embedded in bigger driver structures.

Drivers can pass addresses and ranges in an arbitrary unit, e.g. bytes or pages.

There should be one manager instance per GPU virtual address space.

drm_gpuva_for_each_va_range*drm_gpuva_for_each_va_range* (va__, mgr__, start__, end__)iterate over a range of *drm_gpuvas***Parameters****va__***drm_gpuva* structure to assign to in each iteration step**mgr__***drm_gpuva_manager* to walk over**start__**starting offset, the first *gpuva* will overlap this**end__**ending offset, the last *gpuva* will start before this (but may overlap)**Description**

This iterator walks over all *drm_gpuvas* in the *drm_gpuva_manager* that lie between **start__** and **end__**. It is implemented similarly to *list_for_each()*, but is using the *drm_gpuva_manager*'s internal interval tree to accelerate the search for the starting *drm_gpuva*, and hence isn't safe against removal of elements. It assumes that **end__** is within (or is the upper limit of) the *drm_gpuva_manager*. This iterator does not skip over the *drm_gpuva_manager*'s **kernel_alloc_node**.

drm_gpuva_for_each_va_range_safe*drm_gpuva_for_each_va_range_safe* (va__, next__, mgr__, start__, end__)safely iterate over a range of *drm_gpuvas***Parameters****va__***drm_gpuva* to assign to in each iteration step**next__**another *drm_gpuva* to use as temporary storage

mgr__
 drm_gpuva_manager to walk over

start__
 starting offset, the first gpuva will overlap this

end__
 ending offset, the last gpuva will start before this (but may overlap)

Description

This iterator walks over all *drm_gpuvas* in the *drm_gpuva_manager* that lie between **start__** and **end__**. It is implemented similarly to *list_for_each_safe()*, but is using the *drm_gpuva_manager*'s internal interval tree to accelerate the search for the starting *drm_gpuva*, and hence is safe against removal of elements. It assumes that **end__** is within (or is the upper limit of) the *drm_gpuva_manager*. This iterator does not skip over the *drm_gpuva_manager*'s **kernel_alloc_node**.

drm_gpuva_for_each_va

drm_gpuva_for_each_va (*va__*, *mgr__*)
 iterate over all *drm_gpuvas*

Parameters

va__
 drm_gpuva to assign to in each iteration step

mgr__
 drm_gpuva_manager to walk over

Description

This iterator walks over all *drm_gpuva* structures associated with the given *drm_gpuva_manager*.

drm_gpuva_for_each_va_safe

drm_gpuva_for_each_va_safe (*va__*, *next__*, *mgr__*)
 safely iterate over all *drm_gpuvas*

Parameters

va__
 drm_gpuva to assign to in each iteration step

next__
 another *drm_gpuva* to use as temporary storage

mgr__
 drm_gpuva_manager to walk over

Description

This iterator walks over all *drm_gpuva* structures associated with the given *drm_gpuva_manager*. It is implemented with *list_for_each_entry_safe()*, and hence safe against the removal of elements.

enum **drm_gpuva_op_type**
 GPU VA operation type

Constants

DRM_GPUVA_OP_MAP

the map op type

DRM_GPUVA_OP_REMAP

the remap op type

DRM_GPUVA_OP_UNMAP

the unmap op type

DRM_GPUVA_OP_PREFETCH

the prefetch op type

Description

Operations to alter the GPU VA mappings tracked by the *drm_gpuva_manager*.

struct **drm_gpuva_op_map**

GPU VA map operation

Definition:

```
struct drm_gpuva_op_map {
    struct {
        u64 addr;
        u64 range;
    } va;
    struct {
        u64 offset;
        struct drm_gem_object *obj;
    } gem;
};
```

Members

va

structure containing address and range of a map operation

gem

structure containing the *drm_gem_object* and it's offset

Description

This structure represents a single map operation generated by the DRM GPU VA manager.

struct **drm_gpuva_op_unmap**

GPU VA unmap operation

Definition:

```
struct drm_gpuva_op_unmap {
    struct drm_gpuva *va;
    bool keep;
};
```

Members

va

the *drm_gpuva* to unmap

keep

Indicates whether this *drm_gpuva* is physically contiguous with the original mapping request.

Optionally, if **keep** is set, drivers may keep the actual page table mappings for this *drm_gpuva*, adding the missing page table entries only and update the *drm_gpuva_manager* accordingly.

Description

This structure represents a single unmap operation generated by the DRM GPU VA manager.

struct **drm_gpuva_op_remap**

GPU VA remap operation

Definition:

```
struct drm_gpuva_op_remap {
    struct drm_gpuva_op_map *prev;
    struct drm_gpuva_op_map *next;
    struct drm_gpuva_op_unmap *unmap;
};
```

Members

prev

the preceding part of a split mapping

next

the subsequent part of a split mapping

unmap

the unmap operation for the original existing mapping

Description

This represents a single remap operation generated by the DRM GPU VA manager.

A remap operation is generated when an existing GPU VA mmapping is split up by inserting a new GPU VA mapping or by partially unmapping existent mapping(s), hence it consists of a maximum of two map and one unmap operation.

The **unmap** operation takes care of removing the original existing mapping. **prev** is used to remap the preceding part, **next** the subsequent part.

If either a new mapping's start address is aligned with the start address of the old mapping or the new mapping's end address is aligned with the end address of the old mapping, either **prev** or **next** is NULL.

Note, the reason for a dedicated remap operation, rather than arbitrary unmap and map operations, is to give drivers the chance of extracting driver specific data for creating the new mappings from the unmap operations's *drm_gpuva* structure which typically is embedded in larger driver specific structures.

struct **drm_gpuva_op_prefetch**

GPU VA prefetch operation

Definition:

```
struct drm_gpuva_op_prefetch {
    struct drm_gpuva *va;
};
```

Members**va**

the [drm_gpuva](#) to prefetch

Description

This structure represents a single prefetch operation generated by the DRM GPU VA manager.

struct `drm_gpuva_op`

GPU VA operation

Definition:

```
struct drm_gpuva_op {
    struct list_head entry;
    enum drm_gpuva_op_type op;
    union {
        struct drm_gpuva_op_map map;
        struct drm_gpuva_op_remap remap;
        struct drm_gpuva_op_unmap unmap;
        struct drm_gpuva_op_prefetch prefetch;
    };
};
```

Members**entry**

The `list_head` used to distribute instances of this struct within [drm_gpuva_ops](#).

op

the type of the operation

{`unnamed_union`}

anonymous

map

the map operation

remap

the remap operation

unmap

the unmap operation

prefetch

the prefetch operation

Description

This structure represents a single generic operation.

The particular type of the operation is defined by **op**.

struct **drm_gpuva_ops**

wraps a list of *drm_gpuva_op*

Definition:

```
struct drm_gpuva_ops {  
    struct list_head list;  
};
```

Members

list

the list_head

drm_gpuva_for_each_op

drm_gpuva_for_each_op (op, ops)

iterator to walk over *drm_gpuva_ops*

Parameters

op

drm_gpuva_op to assign in each iteration step

ops

drm_gpuva_ops to walk

Description

This iterator walks over all ops within a given list of operations.

drm_gpuva_for_each_op_safe

drm_gpuva_for_each_op_safe (op, next, ops)

iterator to safely walk over *drm_gpuva_ops*

Parameters

op

drm_gpuva_op to assign in each iteration step

next

next *drm_gpuva_op* to store the next step

ops

drm_gpuva_ops to walk

Description

This iterator walks over all ops within a given list of operations. It is implemented with list_for_each_safe(), so save against removal of elements.

drm_gpuva_for_each_op_from_reverse

drm_gpuva_for_each_op_from_reverse (op, ops)

iterate backwards from the given point

Parameters

op

drm_gpuva_op to assign in each iteration step

ops

drm_gpuva_ops to walk

Description

This iterator walks over all ops within a given list of operations beginning from the given operation in reverse order.

drm_gpuva_first_op

drm_gpuva_first_op (ops)

returns the first *drm_gpuva_op* from *drm_gpuva_ops*

Parameters

ops

the *drm_gpuva_ops* to get the first *drm_gpuva_op* from

drm_gpuva_last_op

drm_gpuva_last_op (ops)

returns the last *drm_gpuva_op* from *drm_gpuva_ops*

Parameters

ops

the *drm_gpuva_ops* to get the last *drm_gpuva_op* from

drm_gpuva_prev_op

drm_gpuva_prev_op (op)

previous *drm_gpuva_op* in the list

Parameters

op

the current *drm_gpuva_op*

drm_gpuva_next_op

drm_gpuva_next_op (op)

next *drm_gpuva_op* in the list

Parameters

op

the current *drm_gpuva_op*

struct drm_gpuva_fn_ops

callbacks for split/merge steps

Definition:

```
struct drm_gpuva_fn_ops {
    struct drm_gpuva_op *(*op_alloc)(void);
    void (*op_free)(struct drm_gpuva_op *op);
};
```

```
int (*sm_step_map)(struct drm_gpuva_op *op, void *priv);
int (*sm_step_remap)(struct drm_gpuva_op *op, void *priv);
int (*sm_step_unmap)(struct drm_gpuva_op *op, void *priv);
};
```

Members

op_alloc

called when the *drm_gpuva_manager* allocates a *struct drm_gpuva_op*

Some drivers may want to embed *struct drm_gpuva_op* into driver specific structures. By implementing this callback drivers can allocate memory accordingly.

This callback is optional.

op_free

called when the *drm_gpuva_manager* frees a *struct drm_gpuva_op*

Some drivers may want to embed *struct drm_gpuva_op* into driver specific structures. By implementing this callback drivers can free the previously allocated memory accordingly.

This callback is optional.

sm_step_map

called from *drm_gpuva_sm_map* to finally insert the mapping once all previous steps were completed

The *priv* pointer matches the one the driver passed to *drm_gpuva_sm_map* or *drm_gpuva_sm_unmap*, respectively.

Can be NULL if *drm_gpuva_sm_map* is used.

sm_step_remap

called from *drm_gpuva_sm_map* and *drm_gpuva_sm_unmap* to split up an existent mapping

This callback is called when existent mapping needs to be split up. This is the case when either a newly requested mapping overlaps or is enclosed by an existent mapping or a partial unmap of an existent mapping is requested.

The *priv* pointer matches the one the driver passed to *drm_gpuva_sm_map* or *drm_gpuva_sm_unmap*, respectively.

Can be NULL if neither *drm_gpuva_sm_map* nor *drm_gpuva_sm_unmap* is used.

sm_step_unmap

called from *drm_gpuva_sm_map* and *drm_gpuva_sm_unmap* to unmap an existent mapping

This callback is called when existent mapping needs to be unmapped. This is the case when either a newly requested mapping encloses an existent mapping or an unmap of an existent mapping is requested.

The *priv* pointer matches the one the driver passed to *drm_gpuva_sm_map* or *drm_gpuva_sm_unmap*, respectively.

Can be NULL if neither *drm_gpuva_sm_map* nor *drm_gpuva_sm_unmap* is used.

Description

This structure defines the callbacks used by *drm_gpuva_sm_map* and *drm_gpuva_sm_unmap* to provide the split/merge steps for map and unmap operations to drivers.


```
void drm_gpuva_manager_init(struct drm_gpuva_manager *mgr, const char *name, u64
                           start_offset, u64 range, u64 reserve_offset, u64
                           reserve_range, const struct drm_gpuva_fn_ops *ops)
    initialize a drm_gpuva_manager
```

Parameters

struct *drm_gpuva_manager* *mgr
pointer to the *drm_gpuva_manager* to initialize

const char *name
the name of the GPU VA space

u64 start_offset
the start offset of the GPU VA space

u64 range
the size of the GPU VA space

u64 reserve_offset
the start of the kernel reserved GPU VA area

u64 reserve_range
the size of the kernel reserved GPU VA area

const struct *drm_gpuva_fn_ops* *ops
drm_gpuva_fn_ops called on *drm_gpuva_sm_map* / *drm_gpuva_sm_unmap*

Description

The *drm_gpuva_manager* must be initialized with this function before use.

Note that **mgr** must be cleared to 0 before calling this function. The given name is expected to be managed by the surrounding driver structures.

```
void drm_gpuva_manager_destroy(struct drm_gpuva_manager *mgr)
    cleanup a drm_gpuva_manager
```

Parameters

struct *drm_gpuva_manager* *mgr
pointer to the *drm_gpuva_manager* to clean up

Description

Note that it is a bug to call this function on a manager that still holds GPU VA mappings.

```
int drm_gpuva_insert(struct drm_gpuva_manager *mgr, struct drm_gpuva *va)
    insert a drm_gpuva
```

Parameters

struct *drm_gpuva_manager* *mgr
the *drm_gpuva_manager* to insert the *drm_gpuva* in

struct *drm_gpuva* *va
the *drm_gpuva* to insert

Description

Insert a *drm_gpuva* with a given address and range into a *drm_gpuva_manager*.

It is safe to use this function using the safe versions of iterating the GPU VA space, such as *drm_gpuva_for_each_va_safe()* and *drm_gpuva_for_each_va_range_safe()*.

Return

0 on success, negative error code on failure.

void **drm_gpuva_remove**(struct *drm_gpuva* *va)
remove a *drm_gpuva*

Parameters

struct drm_gpuva *va
the *drm_gpuva* to remove

Description

This removes the given va from the underlying tree.

It is safe to use this function using the safe versions of iterating the GPU VA space, such as *drm_gpuva_for_each_va_safe()* and *drm_gpuva_for_each_va_range_safe()*.

void **drm_gpuva_link**(struct *drm_gpuva* *va)
link a *drm_gpuva*

Parameters

struct drm_gpuva *va
the *drm_gpuva* to link

Description

This adds the given va to the GPU VA list of the *drm_gem_object* it is associated with.

This function expects the caller to protect the GEM's GPUVA list against concurrent access using the GEMs dma_resv lock.

void **drm_gpuva_unlink**(struct *drm_gpuva* *va)
unlink a *drm_gpuva*

Parameters

struct drm_gpuva *va
the *drm_gpuva* to unlink

Description

This removes the given va from the GPU VA list of the *drm_gem_object* it is associated with.

This function expects the caller to protect the GEM's GPUVA list against concurrent access using the GEMs dma_resv lock.

struct *drm_gpuva* ***drm_gpuva_find_first**(struct *drm_gpuva_manager* *mgr, u64 addr, u64 range)
find the first *drm_gpuva* in the given range

Parameters

struct drm_gpuva_manager *mgr
the *drm_gpuva_manager* to search in

u64 addr
the `drm_gpuvas` address

u64 range
the `drm_gpuvas` range

Return

the first `drm_gpuva` within the given range

struct `drm_gpuva` ***drm_gpuva_find**(struct `drm_gpuva_manager` *mgr, u64 addr, u64 range)
find a `drm_gpuva`

Parameters

struct `drm_gpuva_manager` *mgr
the `drm_gpuva_manager` to search in

u64 addr
the `drm_gpuvas` address

u64 range
the `drm_gpuvas` range

Return

the `drm_gpuva` at a given addr and with a given range

struct `drm_gpuva` ***drm_gpuva_find_prev**(struct `drm_gpuva_manager` *mgr, u64 start)
find the `drm_gpuva` before the given address

Parameters

struct `drm_gpuva_manager` *mgr
the `drm_gpuva_manager` to search in

u64 start
the given GPU VA's start address

Description

Find the adjacent `drm_gpuva` before the GPU VA with given start address.

Note that if there is any free space between the GPU VA mappings no mapping is returned.

Return

a pointer to the found `drm_gpuva` or NULL if none was found

struct `drm_gpuva` ***drm_gpuva_find_next**(struct `drm_gpuva_manager` *mgr, u64 end)
find the `drm_gpuva` after the given address

Parameters

struct `drm_gpuva_manager` *mgr
the `drm_gpuva_manager` to search in

u64 end
the given GPU VA's end address

Description

Find the adjacent `drm_gpuva` after the GPU VA with given end address.

Note that if there is any free space between the GPU VA mappings no mapping is returned.

Return

a pointer to the found *drm_gpuva* or NULL if none was found

bool **drm_gpuva_interval_empty**(struct *drm_gpuva_manager* *mgr, u64 addr, u64 range)
indicate whether a given interval of the VA space is empty

Parameters

struct *drm_gpuva_manager* *mgr
the *drm_gpuva_manager* to check the range for

u64 addr
the start address of the range

u64 range
the range of the interval

Return

true if the interval is empty, false otherwise

void **drm_gpuva_map**(struct *drm_gpuva_manager* *mgr, struct *drm_gpuva* *va, struct
drm_gpuva_op_map *op)
helper to insert a *drm_gpuva* according to a *drm_gpuva_op_map*

Parameters

struct *drm_gpuva_manager* *mgr
the *drm_gpuva_manager*

struct *drm_gpuva* *va
the *drm_gpuva* to insert

struct *drm_gpuva_op_map* *op
the *drm_gpuva_op_map* to initialize **va** with

Description

Initializes the **va** from the **op** and inserts it into the given **mgr**.

void **drm_gpuva_remap**(struct *drm_gpuva* *prev, struct *drm_gpuva* *next, struct
drm_gpuva_op_remap *op)
helper to remap a *drm_gpuva* according to a *drm_gpuva_op_remap*

Parameters

struct *drm_gpuva* *prev
the *drm_gpuva* to remap when keeping the start of a mapping

struct *drm_gpuva* *next
the *drm_gpuva* to remap when keeping the end of a mapping

struct *drm_gpuva_op_remap* *op
the *drm_gpuva_op_remap* to initialize **prev** and **next** with

Description

Removes the currently mapped *drm_gpuva* and remaps it using **prev** and/or **next**.

void **drm_gpuva_unmap**(struct *drm_gpuva_op_unmap* *op)
 helper to remove a *drm_gpuva* according to a *drm_gpuva_op_unmap*

Parameters

struct *drm_gpuva_op_unmap* *op
 the *drm_gpuva_op_unmap* specifying the *drm_gpuva* to remove

Description

Removes the *drm_gpuva* associated with the *drm_gpuva_op_unmap*.

int **drm_gpuva_sm_map**(struct *drm_gpuva_manager* *mgr, void *priv, u64 req_addr, u64 req_range, struct *drm_gem_object* *req_obj, u64 req_offset)
 creates the *drm_gpuva_op* split/merge steps

Parameters

struct *drm_gpuva_manager* *mgr
 the *drm_gpuva_manager* representing the GPU VA space

void *priv
 pointer to a driver private data structure

u64 req_addr
 the start address of the new mapping

u64 req_range
 the range of the new mapping

struct *drm_gem_object* *req_obj
 the *drm_gem_object* to map

u64 req_offset
 the offset within the *drm_gem_object*

Description

This function iterates the given range of the GPU VA space. It utilizes the *drm_gpuva_fn_ops* to call back into the driver providing the split and merge steps.

Drivers may use these callbacks to update the GPU VA space right away within the callback. In case the driver decides to copy and store the operations for later processing neither this function nor *drm_gpuva_sm_unmap* is allowed to be called before the *drm_gpuva_manager*'s view of the GPU VA space was updated with the previous set of operations. To update the *drm_gpuva_manager*'s view of the GPU VA space *drm_gpuva_insert()*, *drm_gpuva_destroy_locked()* and/or *drm_gpuva_destroy_unlocked()* should be used.

A sequence of callbacks can contain map, unmap and remap operations, but the sequence of callbacks might also be empty if no operation is required, e.g. if the requested mapping already exists in the exact same way.

There can be an arbitrary amount of unmap operations, a maximum of two remap operations and a single map operation. The latter one represents the original map operation requested by the caller.

Return

0 on success or a negative error code

int **drm_gpuva_sm_unmap**(struct *drm_gpuva_manager* *mgr, void *priv, u64 req_addr, u64 req_range)

creates the *drm_gpuva_ops* to split on unmap

Parameters

struct drm_gpuva_manager *mgr

the *drm_gpuva_manager* representing the GPU VA space

void *priv

pointer to a driver private data structure

u64 req_addr

the start address of the range to unmap

u64 req_range

the range of the mappings to unmap

Description

This function iterates the given range of the GPU VA space. It utilizes the *drm_gpuva_fn_ops* to call back into the driver providing the operations to unmap and, if required, split existent mappings.

Drivers may use these callbacks to update the GPU VA space right away within the callback. In case the driver decides to copy and store the operations for later processing neither this function nor *drm_gpuva_sm_map* is allowed to be called before the *drm_gpuva_manager*'s view of the GPU VA space was updated with the previous set of operations. To update the *drm_gpuva_manager*'s view of the GPU VA space *drm_gpuva_insert()*, *drm_gpuva_destroy_locked()* and/or *drm_gpuva_destroy_unlocked()* should be used.

A sequence of callbacks can contain unmap and remap operations, depending on whether there are actual overlapping mappings to split.

There can be an arbitrary amount of unmap operations and a maximum of two remap operations.

Return

0 on success or a negative error code

struct *drm_gpuva_ops* ***drm_gpuva_sm_map_ops_create**(struct *drm_gpuva_manager* *mgr, u64 req_addr, u64 req_range, struct *drm_gem_object* *req_obj, u64 req_offset)

creates the *drm_gpuva_ops* to split and merge

Parameters

struct drm_gpuva_manager *mgr

the *drm_gpuva_manager* representing the GPU VA space

u64 req_addr

the start address of the new mapping

u64 req_range

the range of the new mapping

struct drm_gem_object *req_obj

the *drm_gem_object* to map

u64 req_offset

the offset within the *drm_gem_object*

Description

This function creates a list of operations to perform splitting and merging of existent mapping(s) with the newly requested one.

The list can be iterated with *drm_gpuva_for_each_op* and must be processed in the given order. It can contain map, unmap and remap operations, but it also can be empty if no operation is required, e.g. if the requested mapping already exists in the exact same way.

There can be an arbitrary amount of unmap operations, a maximum of two remap operations and a single map operation. The latter one represents the original map operation requested by the caller.

Note that before calling this function again with another mapping request it is necessary to update the *drm_gpuva_manager*'s view of the GPU VA space. The previously obtained operations must be either processed or abandoned. To update the *drm_gpuva_manager*'s view of the GPU VA space *drm_gpuva_insert()*, *drm_gpuva_destroy_locked()* and/or *drm_gpuva_destroy_unlocked()* should be used.

After the caller finished processing the returned *drm_gpuva_ops*, they must be freed with *drm_gpuva_ops_free*.

Return

a pointer to the *drm_gpuva_ops* on success, an ERR_PTR on failure

```
struct drm_gpuva_ops *drm_gpuva_sm_unmap_ops_create(struct drm_gpuva_manager *mgr,  
                                                    u64 req_addr, u64 req_range)
```

creates the *drm_gpuva_ops* to split on unmap

Parameters

```
struct drm_gpuva_manager *mgr
```

the *drm_gpuva_manager* representing the GPU VA space

```
u64 req_addr
```

the start address of the range to unmap

```
u64 req_range
```

the range of the mappings to unmap

Description

This function creates a list of operations to perform unmapping and, if required, splitting of the mappings overlapping the unmap range.

The list can be iterated with *drm_gpuva_for_each_op* and must be processed in the given order. It can contain unmap and remap operations, depending on whether there are actual overlapping mappings to split.

There can be an arbitrary amount of unmap operations and a maximum of two remap operations.

Note that before calling this function again with another range to unmap it is necessary to update the *drm_gpuva_manager*'s view of the GPU VA space. The previously obtained operations must be processed or abandoned. To update the *drm_gpuva_manager*'s view of the GPU VA space *drm_gpuva_insert()*, *drm_gpuva_destroy_locked()* and/or *drm_gpuva_destroy_unlocked()* should be used.

After the caller finished processing the returned *drm_gpuva_ops*, they must be freed with *drm_gpuva_ops_free*.

Return

a pointer to the *drm_gpuva_ops* on success, an ERR_PTR on failure

struct *drm_gpuva_ops* ***drm_gpuva_prefetch_ops_create**(struct *drm_gpuva_manager* *mgr,
u64 addr, u64 range)

creates the *drm_gpuva_ops* to prefetch

Parameters

struct *drm_gpuva_manager* *mgr
the *drm_gpuva_manager* representing the GPU VA space

u64 addr
the start address of the range to prefetch

u64 range
the range of the mappings to prefetch

Description

This function creates a list of operations to perform prefetching.

The list can be iterated with *drm_gpuva_for_each_op* and must be processed in the given order. It can contain prefetch operations.

There can be an arbitrary amount of prefetch operations.

After the caller finished processing the returned *drm_gpuva_ops*, they must be freed with *drm_gpuva_ops_free*.

Return

a pointer to the *drm_gpuva_ops* on success, an ERR_PTR on failure

struct *drm_gpuva_ops* ***drm_gpuva_gem_unmap_ops_create**(struct *drm_gpuva_manager* *mgr,
struct *drm_gem_object* *obj)

creates the *drm_gpuva_ops* to unmap a GEM

Parameters

struct *drm_gpuva_manager* *mgr
the *drm_gpuva_manager* representing the GPU VA space

struct *drm_gem_object* *obj
the *drm_gem_object* to unmap

Description

This function creates a list of operations to perform unmapping for every GPUVA attached to a GEM.

The list can be iterated with *drm_gpuva_for_each_op* and consists out of an arbitrary amount of unmap operations.

After the caller finished processing the returned *drm_gpuva_ops*, they must be freed with *drm_gpuva_ops_free*.

It is the callers responsibility to protect the GEMs GPUVA list against concurrent access using the GEMs dma_resv lock.

Return

a pointer to the *drm_gpuva_ops* on success, an ERR_PTR on failure

void **drm_gpuva_ops_free**(struct *drm_gpuva_manager* *mgr, struct *drm_gpuva_ops* *ops)
free the given *drm_gpuva_ops*

Parameters

struct drm_gpuva_manager *mgr
the *drm_gpuva_manager* the ops were created for

struct drm_gpuva_ops *ops
the *drm_gpuva_ops* to free

Description

Frees the given *drm_gpuva_ops* structure including all the ops associated with it.

3.7 DRM Buddy Allocator

3.7.1 DRM Buddy Function References

int **drm_buddy_init**(struct drm_buddy *mm, u64 size, u64 chunk_size)
init memory manager

Parameters

struct drm_buddy *mm
DRM buddy manager to initialize

u64 size
size in bytes to manage

u64 chunk_size
minimum page size in bytes for our allocations

Description

Initializes the memory manager and its resources.

Return

0 on success, error code on failure.

void **drm_buddy_fini**(struct drm_buddy *mm)
tear down the memory manager

Parameters

struct drm_buddy *mm
DRM buddy manager to free

Description

Cleanup memory manager resources and the freelist

struct drm_buddy_block ***drm_get_buddy**(struct drm_buddy_block *block)
get buddy address

Parameters

struct drm_buddy_block *block
DRM buddy block

Description

Returns the corresponding buddy block for **block**, or NULL if this is a root block and can't be merged further. Requires some kind of locking to protect against any concurrent allocate and free operations.

void **drm_buddy_free_block**(struct drm_buddy *mm, struct drm_buddy_block *block)
free a block

Parameters

struct drm_buddy *mm
DRM buddy manager

struct drm_buddy_block *block
block to be freed

void **drm_buddy_free_list**(struct drm_buddy *mm, struct list_head *objects)
free blocks

Parameters

struct drm_buddy *mm
DRM buddy manager

struct list_head *objects
input list head to free blocks

int **drm_buddy_block_trim**(struct drm_buddy *mm, u64 new_size, struct list_head *blocks)
free unused pages

Parameters

struct drm_buddy *mm
DRM buddy manager

u64 new_size
original size requested

struct list_head *blocks
Input and output list of allocated blocks. MUST contain single block as input to be trimmed. On success will contain the newly allocated blocks making up the **new_size**. Blocks always appear in ascending order

Description

For contiguous allocation, we round up the size to the nearest power of two value, drivers consume *actual* size, so remaining portions are unused and can be optionally freed with this function

Return

0 on success, error code on failure.

int **drm_buddy_alloc_blocks**(struct drm_buddy *mm, u64 start, u64 end, u64 size, u64 min_page_size, struct list_head *blocks, unsigned long flags)
allocate power-of-two blocks

Parameters

struct drm_buddy *mm
DRM buddy manager to allocate from

u64 start
start of the allowed range for this block

u64 end
end of the allowed range for this block

u64 size
size of the allocation

u64 min_page_size
alignment of the allocation

struct list_head *blocks
output list head to add allocated blocks

unsigned long flags
DRM_BUDDY_*_ALLOCATION flags

Description

alloc_range_bias() called on range limitations, which traverses the tree and returns the desired block.

alloc_from_freelist() called when *no* range restrictions are enforced, which picks the block from the freelist.

Return

0 on success, error code on failure.

void **drm_buddy_block_print**(struct drm_buddy *mm, struct drm_buddy_block *block, struct *drm_printer* *p)
print block information

Parameters

struct drm_buddy *mm
DRM buddy manager

struct drm_buddy_block *block
DRM buddy block

struct drm_printer *p
DRM printer to use

void **drm_buddy_print**(struct drm_buddy *mm, struct *drm_printer* *p)
print allocator state

Parameters

struct drm_buddy *mm
DRM buddy manager

struct drm_printer *p
DRM printer to use

3.8 DRM Cache Handling and Fast WC memcpy()

void drm_clflush_pages(struct page *pages[], unsigned long num_pages)
Flush dcache lines of a set of pages.

Parameters

struct page *pages[]
List of pages to be flushed.

unsigned long num_pages
Number of pages in the array.

Description

Flush every data cache line entry that points to an address belonging to a page in the array.

void drm_clflush_sg(struct sg_table *st)
Flush dcache lines pointing to a scatter-gather.

Parameters

struct sg_table *st
struct sg_table.

Description

Flush every data cache line entry that points to an address in the sg.

void drm_clflush_virt_range(void *addr, unsigned long length)
Flush dcache lines of a region

Parameters

void *addr
Initial kernel memory address.

unsigned long length
Region size.

Description

Flush every data cache line entry that points to an address in the region requested.

void drm_memcpy_from_wc(struct iosys_map *dst, const struct iosys_map *src, unsigned long len)
Perform the fastest available memcpy from a source that may be WC.

Parameters

struct iosys_map *dst
The destination pointer

const struct iosys_map *src
The source pointer

unsigned long len

The size of the area to transfer in bytes

Description

Tries an arch optimized memcpy for prefetching reading out of a WC region, and if no such beast is available, falls back to a normal memcpy.

3.9 DRM Sync Objects

DRM synchronisation objects (syncobj, see struct *drm_syncobj*) provide a container for a synchronization primitive which can be used by userspace to explicitly synchronize GPU commands, can be shared between userspace processes, and can be shared between different DRM drivers. Their primary use-case is to implement Vulkan fences and semaphores. The syncobj userspace API provides ioctls for several operations:

- Creation and destruction of syncobjs
- Import and export of syncobjs to/from a syncobj file descriptor
- Import and export a syncobj's underlying fence to/from a sync file
- Reset a syncobj (set its fence to NULL)
- Signal a syncobj (set a trivially signaled fence)
- Wait for a syncobj's fence to appear and be signaled

The syncobj userspace API also provides operations to manipulate a syncobj in terms of a timeline of struct *dma_fence_chain* rather than a single struct *dma_fence*, through the following operations:

- Signal a given point on the timeline
- Wait for a given point to appear and/or be signaled
- Import and export from/to a given point of a timeline

At its core, a syncobj is simply a wrapper around a pointer to a struct *dma_fence* which may be NULL. When a syncobj is first created, its pointer is either NULL or a pointer to an already signaled fence depending on whether the *DRM_SYNCOBJ_CREATE_SIGNALED* flag is passed to *DRM_IOCTL_SYNCOBJ_CREATE*.

If the syncobj is considered as a binary (its state is either signaled or unsignaled) primitive, when GPU work is enqueued in a DRM driver to signal the syncobj, the syncobj's fence is replaced with a fence which will be signaled by the completion of that work. If the syncobj is considered as a timeline primitive, when GPU work is enqueued in a DRM driver to signal the a given point of the syncobj, a new struct *dma_fence_chain* pointing to the DRM driver's fence and also pointing to the previous fence that was in the syncobj. The new struct *dma_fence_chain* fence replace the syncobj's fence and will be signaled by completion of the DRM driver's work and also any work associated with the fence previously in the syncobj.

When GPU work which waits on a syncobj is enqueued in a DRM driver, at the time the work is enqueued, it waits on the syncobj's fence before submitting the work to hardware. That fence is either :

- The syncobj's current fence if the syncobj is considered as a binary primitive.

- The struct `dma_fence` associated with a given point if the syncobj is considered as a timeline primitive.

If the syncobj's fence is NULL or not present in the syncobj's timeline, the enqueue operation is expected to fail.

With binary syncobj, all manipulation of the syncobj's fence happens in terms of the current fence at the time the ioctl is called by userspace regardless of whether that operation is an immediate host-side operation (signal or reset) or an operation which is enqueued in some driver queue. `DRM_IOCTL_SYNCOBJ_RESET` and `DRM_IOCTL_SYNCOBJ_SIGNAL` can be used to manipulate a syncobj from the host by resetting its pointer to NULL or setting its pointer to a fence which is already signaled.

With a timeline syncobj, all manipulation of the syncobj's fence happens in terms of a u64 value referring to point in the timeline. See `dma_fence_chain_find_seqno()` to see how a given point is found in the timeline.

Note that applications should be careful to always use timeline set of ioctl() when dealing with syncobj considered as timeline. Using a binary set of ioctl() with a syncobj considered as timeline could result incorrect synchronization. The use of binary syncobj is supported through the timeline set of ioctl() by using a point value of 0, this will reproduce the behavior of the binary set of ioctl() (for example replace the syncobj's fence when signaling).

3.9.1 Host-side wait on syncobjs

`DRM_IOCTL_SYNCOBJ_WAIT` takes an array of syncobj handles and does a host-side wait on all of the syncobj fences simultaneously. If `DRM_SYNCOBJ_WAIT_FLAGS_WAIT_ALL` is set, the wait ioctl will wait on all of the syncobj fences to be signaled before it returns. Otherwise, it returns once at least one syncobj fence has been signaled and the index of a signaled fence is written back to the client.

Unlike the enqueued GPU work dependencies which fail if they see a NULL fence in a syncobj, if `DRM_SYNCOBJ_WAIT_FLAGS_WAIT_FOR_SUBMIT` is set, the host-side wait will first wait for the syncobj to receive a non-NULL fence and then wait on that fence. If `DRM_SYNCOBJ_WAIT_FLAGS_WAIT_FOR_SUBMIT` is not set and any one of the syncobjs in the array has a NULL fence, `-EINVAL` will be returned. Assuming the syncobj starts off with a NULL fence, this allows a client to do a host wait in one thread (or process) which waits on GPU work submitted in another thread (or process) without having to manually synchronize between the two. This requirement is inherited from the Vulkan fence API.

Similarly, `DRM_IOCTL_SYNCOBJ_TIMELINE_WAIT` takes an array of syncobj handles as well as an array of u64 points and does a host-side wait on all of syncobj fences at the given points simultaneously.

`DRM_IOCTL_SYNCOBJ_TIMELINE_WAIT` also adds the ability to wait for a given fence to materialize on the timeline without waiting for the fence to be signaled by using the `DRM_SYNCOBJ_WAIT_FLAGS_WAIT_AVAILABLE` flag. This requirement is inherited from the wait-before-signal behavior required by the Vulkan timeline semaphore API.

Alternatively, `DRM_IOCTL_SYNCOBJ_EVENTFD` can be used to wait without blocking: an eventfd will be signaled when the syncobj is. This is useful to integrate the wait in an event loop.

3.9.2 Import/export of syncobjs

DRM_IOCTL_SYNCOBJ_FD_TO_HANDLE and DRM_IOCTL_SYNCOBJ_HANDLE_TO_FD provide two mechanisms for import/export of syncobjs.

The first lets the client import or export an entire syncobj to a file descriptor. These fd's are opaque and have no other use case, except passing the syncobj between processes. All exported file descriptors and any syncobj handles created as a result of importing those file descriptors own a reference to the same underlying struct *drm_syncobj* and the syncobj can be used persistently across all the processes with which it is shared. The syncobj is freed only once the last reference is dropped. Unlike dma-buf, importing a syncobj creates a new handle (with its own reference) for every import instead of de-duplicating. The primary use-case of this persistent import/export is for shared Vulkan fences and semaphores.

The second import/export mechanism, which is indicated by DRM_SYNCOBJ_FD_TO_HANDLE_FLAGS_IMPORT_SYNC_FILE or DRM_SYNCOBJ_HANDLE_TO_FD_FLAGS_EXPORT_SYNC_FILE lets the client import/export the syncobj's current fence from/to a sync_file. When a syncobj is exported to a sync file, that sync file wraps the syncobj's fence at the time of export and any later signal or reset operations on the syncobj will not affect the exported sync file. When a sync file is imported into a syncobj, the syncobj's fence is set to the fence wrapped by that sync file. Because sync files are immutable, resetting or signaling the syncobj will not affect any sync files whose fences have been imported into the syncobj.

3.9.3 Import/export of timeline points in timeline syncobjs

DRM_IOCTL_SYNCOBJ_TRANSFER provides a mechanism to transfer a struct dma_fence_chain of a syncobj at a given u64 point to another u64 point into another syncobj.

Note that if you want to transfer a struct dma_fence_chain from a given point on a timeline syncobj from/into a binary syncobj, you can use the point 0 to mean take/replace the fence in the syncobj.

struct **drm_syncobj**
sync object.

Definition:

```
struct drm_syncobj {
    struct kref refcount;
    struct dma_fence __rcu *fence;
    struct list_head cb_list;
    struct list_head ev_fd_list;
    spinlock_t lock;
    struct file *file;
};
```

Members

refcount

Reference count of this object.

fence

NULL or a pointer to the fence bound to this object.

This field should not be used directly. Use `drm_syncobj_fence_get()` and `drm_syncobj_replace_fence()` instead.

cb_list

List of callbacks to call when the fence gets replaced.

ev_fd_list

List of registered eventfd.

lock

Protects `cb_list` and `ev_fd_list`, and write-locks `fence`.

file

A file backing for this syncobj.

Description

This structure defines a generic sync object which wraps a `dma_fence`.

void **drm_syncobj_get**(struct *drm_syncobj* *obj)

acquire a syncobj reference

Parameters

struct drm_syncobj *obj

sync object

Description

This acquires an additional reference to **obj**. It is illegal to call this without already holding a reference. No locks required.

void **drm_syncobj_put**(struct *drm_syncobj* *obj)

release a reference to a sync object.

Parameters

struct drm_syncobj *obj

sync object.

struct `dma_fence` ***drm_syncobj_fence_get**(struct *drm_syncobj* *syncobj)

get a reference to a fence in a sync object

Parameters

struct drm_syncobj *syncobj

sync object.

Description

This acquires additional reference to `drm_syncobj.fence` contained in **obj**, if not NULL. It is illegal to call this without already holding a reference. No locks required.

Return

Either the fence of **obj** or NULL if there's none.

struct *drm_syncobj* ***drm_syncobj_find**(struct *drm_file* *file_private, u32 handle)

lookup and reference a sync object.

Parameters

struct drm_file *file_private

drm file private pointer

u32 handle

sync object handle to lookup.

Description

Returns a reference to the syncobj pointed to by handle or NULL. The reference must be released by calling [drm_syncobj_put\(\)](#).

void **drm_syncobj_add_point**(struct [drm_syncobj](#) *syncobj, struct dma_fence_chain *chain, struct dma_fence *fence, uint64_t point)

add new timeline point to the syncobj

Parameters

struct drm_syncobj *syncobj

sync object to add timeline point to

struct dma_fence_chain *chain

chain node to use to add the point

struct dma_fence *fence

fence to encapsulate in the chain node

uint64_t point

sequence number to use for the point

Description

Add the chain node as new timeline point to the syncobj.

void **drm_syncobj_replace_fence**(struct [drm_syncobj](#) *syncobj, struct dma_fence *fence)
replace fence in a sync object.

Parameters

struct drm_syncobj *syncobj

Sync object to replace fence in

struct dma_fence *fence

fence to install in sync file.

Description

This replaces the fence on a sync object.

int **drm_syncobj_find_fence**(struct [drm_file](#) *file_private, u32 handle, u64 point, u64 flags, struct dma_fence **fence)

lookup and reference the fence in a sync object

Parameters

struct drm_file *file_private

drm file private pointer

u32 handle

sync object handle to lookup.

u64 point

timeline point

u64 flags

DRM_SYNCOBJ_WAIT_FLAGS_WAIT_FOR_SUBMIT or not

struct dma_fence **fence

out parameter for the fence

Description

This is just a convenience function that combines [drm_syncobj_find\(\)](#) and [drm_syncobj_fence_get\(\)](#).

Returns 0 on success or a negative error value on failure. On success **fence** contains a reference to the fence, which must be released by calling [dma_fence_put\(\)](#).

void **drm_syncobj_free**(struct [kref](#) *kref)

free a sync object.

Parameters

struct [kref](#) *kref

kref to free.

Description

Only to be called from [kref_put](#) in [drm_syncobj_put](#).

int **drm_syncobj_create**(struct [drm_syncobj](#) **out_syncobj, uint32_t flags, struct dma_fence *fence)

create a new syncobj

Parameters

struct [drm_syncobj](#) **out_syncobj

returned syncobj

uint32_t flags

DRM_SYNCOBJ_* flags

struct dma_fence *fence

if non-NULL, the syncobj will represent this fence

Description

This is the first function to create a sync object. After creating, drivers probably want to make it available to userspace, either through [drm_syncobj_get_handle\(\)](#) or [drm_syncobj_get_fd\(\)](#).

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

int **drm_syncobj_get_handle**(struct [drm_file](#) *file_private, struct [drm_syncobj](#) *syncobj, u32 *handle)

get a handle from a syncobj

Parameters

struct [drm_file](#) *file_private

drm file private pointer

struct [drm_syncobj](#) *syncobj

Sync object to export

u32 *handle

out parameter with the new handle

Description

Exports a sync object created with `drm_syncobj_create()` as a handle on **file_private** to userspace.

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

int **drm_syncobj_get_fd**(struct *drm_syncobj* *syncobj, int *p_fd)
get a file descriptor from a syncobj

Parameters

struct drm_syncobj *syncobj
Sync object to export

int *p_fd
out parameter with the new file descriptor

Description

Exports a sync object created with `drm_syncobj_create()` as a file descriptor.

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

signed long **drm_timeout_abs_to_jiffies**(int64_t timeout_nsec)
calculate jiffies timeout from absolute value

Parameters

int64_t timeout_nsec
timeout nsec component in ns, 0 for poll

Description

Calculate the timeout in jiffies from an absolute time in sec/nsec.

3.10 DRM Execution context

This component mainly abstracts the retry loop necessary for locking multiple GEM objects while preparing hardware operations (e.g. command submissions, page table updates etc..).

If a contention is detected while locking a GEM object the cleanup procedure unlocks all previously locked GEM objects and locks the contended one first before locking any further objects.

After an object is locked fences slots can optionally be reserved on the dma_resv object inside the GEM object.

A typical usage pattern should look like this:

```
struct drm_gem_object *obj;
struct drm_exec exec;
unsigned long index;
int ret;

drm_exec_init(&exec, DRM_EXEC_INTERRUPTIBLE_WAIT);
drm_exec_until_all_locked(&exec) {
    ret = drm_exec_prepare_obj(&exec, boA, 1);
    drm_exec_retry_on_contention(&exec);
}
```

```
        if (ret)
            goto error;

        ret = drm_exec_prepare_obj(&exec, boB, 1);
        drm_exec_retry_on_contention(&exec);
        if (ret)
            goto error;
    }

    drm_exec_for_each_locked_object(&exec, index, obj) {
        dma_resv_add_fence(obj->resv, fence, DMA_RESV_USAGE_READ);
        ...
    }
    drm_exec_fini(&exec);
```

See struct `dma_exec` for more details.

struct **drm_exec**

Execution context

Definition:

```
struct drm_exec {
    uint32_t flags;
    struct ww_acquire_ctx    ticket;
    unsigned int             num_objects;
    unsigned int             max_objects;
    struct drm_gem_object    **objects;
    struct drm_gem_object    *contended;
    struct drm_gem_object    *prelocked;
};
```

Members

flags

Flags to control locking behavior

ticket

WW ticket used for acquiring locks

num_objects

number of objects locked

max_objects

maximum objects in array

objects

array of the locked objects

contended

contended GEM object we backed off for

prelocked

already locked GEM object due to contention

struct *drm_gem_object* ***drm_exec_obj** (struct *drm_exec* *exec, unsigned long index)

Return the object for a give drm_exec index

Parameters

struct drm_exec *exec

Pointer to the drm_exec context

unsigned long index

The index.

Return

Pointer to the locked object corresponding to **index** if index is within the number of locked objects. NULL otherwise.

drm_exec_for_each_locked_object

drm_exec_for_each_locked_object (exec, index, obj)

iterate over all the locked objects

Parameters

exec

drm_exec object

index

unsigned long index for the iteration

obj

the current GEM object

Description

Iterate over all the locked GEM objects inside the drm_exec object.

drm_exec_for_each_locked_object_reverse

drm_exec_for_each_locked_object_reverse (exec, index, obj)

iterate over all the locked objects in reverse locking order

Parameters

exec

drm_exec object

index

unsigned long index for the iteration

obj

the current GEM object

Description

Iterate over all the locked GEM objects inside the drm_exec object in reverse locking order. Note that **index** may go below zero and wrap, but that will be caught by *drm_exec_obj()*, returning a NULL object.

drm_exec_until_all_locked

drm_exec_until_all_locked (exec)

loop until all GEM objects are locked

Parameters

exec

drm_exec object

Description

Core functionality of the `drm_exec` object. Loops until all GEM objects are locked and no more contention exists. At the beginning of the loop it is guaranteed that no GEM object is locked.

Since labels can't be defined local to the loops body we use a jump pointer to make sure that the retry is only used from within the loops body.

drm_exec_retry_on_contention

drm_exec_retry_on_contention (exec)

restart the loop to grap all locks

Parameters

exec

drm_exec object

Description

Control flow helper to continue when a contention was detected and we need to clean up and re-start the loop to prepare all GEM objects.

bool **drm_exec_is_contended**(struct *drm_exec* *exec)

check for contention

Parameters

struct drm_exec *exec

drm_exec object

Description

Returns true if the `drm_exec` object has run into some contention while locking a GEM object and needs to clean up.

void **drm_exec_init**(struct *drm_exec* *exec, uint32_t flags)

initialize a `drm_exec` object

Parameters

struct drm_exec *exec

the `drm_exec` object to initialize

uint32_t flags

controls locking behavior, see `DRM_EXEC_*` defines

Description

Initialize the object and make sure that we can track locked objects.

void **drm_exec_fini**(struct *drm_exec* *exec)
finalize a drm_exec object

Parameters

struct drm_exec *exec
the drm_exec object to finalize

Description

Unlock all locked objects, drop the references to objects and free all memory used for tracking the state.

bool **drm_exec_cleanup**(struct *drm_exec* *exec)
cleanup when contention is detected

Parameters

struct drm_exec *exec
the drm_exec object to cleanup

Description

Cleanup the current state and return true if we should stay inside the retry loop, false if there wasn't any contention detected and we can keep the objects locked.

int **drm_exec_lock_obj**(struct *drm_exec* *exec, struct *drm_gem_object* *obj)
lock a GEM object for use

Parameters

struct drm_exec *exec
the drm_exec object with the state

struct drm_gem_object *obj
the GEM object to lock

Description

Lock a GEM object for use and grab a reference to it.

Return

-EDEADLK if a contention is detected, -EALREADY when object is already locked (can be suppressed by setting the DRM_EXEC_IGNORE_DUPLICATES flag), -ENOMEM when memory allocation failed and zero for success.

void **drm_exec_unlock_obj**(struct *drm_exec* *exec, struct *drm_gem_object* *obj)
unlock a GEM object in this exec context

Parameters

struct drm_exec *exec
the drm_exec object with the state

struct drm_gem_object *obj
the GEM object to unlock

Description

Unlock the GEM object and remove it from the collection of locked objects. Should only be used to unlock the most recently locked objects. It's not time efficient to unlock objects locked long ago.

int **drm_exec_prepare_obj**(struct *drm_exec* *exec, struct *drm_gem_object* *obj, unsigned int num_fences)

prepare a GEM object for use

Parameters

struct drm_exec *exec
the *drm_exec* object with the state

struct drm_gem_object *obj
the GEM object to prepare

unsigned int num_fences
how many fences to reserve

Description

Prepare a GEM object for use by locking it and reserving fence slots.

Return

-EDEADLK if a contention is detected, -EALREADY when object is already locked, -ENOMEM when memory allocation failed and zero for success.

int **drm_exec_prepare_array**(struct *drm_exec* *exec, struct *drm_gem_object* **objects, unsigned int num_objects, unsigned int num_fences)

helper to prepare an array of objects

Parameters

struct drm_exec *exec
the *drm_exec* object with the state

struct drm_gem_object **objects
array of GEM object to prepare

unsigned int num_objects
number of GEM objects in the array

unsigned int num_fences
number of fences to reserve on each GEM object

Description

Prepares all GEM objects in an array, aborts on first error. Reserves **num_fences** on each GEM object after locking it.

Return

-EDEADLOCK on contention, -EALREADY when object is already locked, -ENOMEM when memory allocation failed and zero for success.

3.11 GPU Scheduler

3.11.1 Overview

The GPU scheduler provides entities which allow userspace to push jobs into software queues which are then scheduled on a hardware run queue. The software queues have a priority among them. The scheduler selects the entities from the run queue using a FIFO. The scheduler provides dependency handling features among jobs. The driver is supposed to provide callback functions for backend operations to the scheduler like submitting a job to hardware run queue, returning the dependencies of a job etc.

The organisation of the scheduler is the following:

1. Each hw run queue has one scheduler
2. Each scheduler has multiple run queues with different priorities (e.g., HIGH_HW, HIGH_SW, KERNEL, NORMAL)
3. Each scheduler run queue has a queue of entities to schedule
4. Entities themselves maintain a queue of jobs that will be scheduled on the hardware.

The jobs in a entity are always scheduled in the order that they were pushed.

Note that once a job was taken from the entities queue and pushed to the hardware, i.e. the pending queue, the entity must not be referenced anymore through the jobs entity pointer.

3.11.2 Scheduler Function References

DRM_SCHED_FENCE_DONT_PIPELINE

DRM_SCHED_FENCE_DONT_PIPELINE ()

Prefer dependency pipelining

Parameters

Description

Setting this flag on a scheduler fence prevents pipelining of jobs depending on this fence. In other words we always insert a full CPU round trip before dependent jobs are pushed to the hw queue.

DRM_SCHED_FENCE_FLAG_HAS_DEADLINE_BIT

DRM_SCHED_FENCE_FLAG_HAS_DEADLINE_BIT ()

A fence deadline hint has been set

Parameters

Description

Because we could have a deadline hint can be set before the backing hw fence is created, we need to keep track of whether a deadline has already been set.

struct `drm_sched_entity`

A wrapper around a job queue (typically attached to the DRM file_priv).

Definition:

```
struct drm_sched_entity {
    struct list_head          list;
    struct drm_sched_rq      *rq;
    struct drm_gpu_scheduler **sched_list;
    unsigned int             num_sched_list;
    enum drm_sched_priority  priority;
    spinlock_t               rq_lock;
    struct spsc_queue        job_queue;
    atomic_t                 fence_seq;
    uint64_t                 fence_context;
    struct dma_fence         *dependency;
    struct dma_fence_cb      cb;
    atomic_t                 *guilty;
    struct dma_fence         __rcu *last_scheduled;
    struct task_struct       *last_user;
    bool                     stopped;
    struct completion        entity_idle;
    ktime_t                 oldest_job_waiting;
    struct rb_node           rb_tree_node;
};
```

Members**list**

Used to append this struct to the list of entities in the runqueue **rq** under *drm_sched_rq.entities*.

Protected by *drm_sched_rq.lock* of **rq**.

rq

Runqueue on which this entity is currently scheduled.

FIXME: Locking is very unclear for this. Writers are protected by **rq_lock**, but readers are generally lockless and seem to just race with not even a READ_ONCE.

sched_list

A list of schedulers (*struct drm_gpu_scheduler*). Jobs from this entity can be scheduled on any scheduler on this list.

This can be modified by calling *drm_sched_entity_modify_sched()*. Locking is entirely up to the driver, see the above function for more details.

This will be set to NULL if num_sched_list equals 1 and **rq** has been set already.

FIXME: This means priority changes through *drm_sched_entity_set_priority()* will be lost henceforth in this case.

num_sched_list

Number of drm_gpu_schedulers in the **sched_list**.

priority

Priority of the entity. This can be modified by calling *drm_sched_entity_set_priority()*.

Protected by `rq_lock`.

rq_lock

Lock to modify the runqueue to which this entity belongs.

job_queue

the list of jobs of this entity.

fence_seq

A linearly increasing seqno incremented with each new `drm_sched_fence` which is part of the entity.

FIXME: Callers of `drm_sched_job_arm()` need to ensure correct locking, this doesn't need to be atomic.

fence_context

A unique context for all the fences which belong to this entity. The `drm_sched_fence.scheduled` uses the `fence_context` but `drm_sched_fence.finished` uses `fence_context + 1`.

dependency

The dependency fence of the job which is on the top of the job queue.

cb

Callback for the dependency fence above.

guilty

Points to entities' guilty.

last_scheduled

Points to the finished fence of the last scheduled job. Only written by the scheduler thread, can be accessed locklessly from `drm_sched_job_arm()` iff the queue is empty.

last_user

last group leader pushing a job into the entity.

stopped

Marks the entity as removed from `rq` and destined for termination. This is set by calling `drm_sched_entity_flush()` and by `drm_sched_fini()`.

entity_idle

Signals when entity is not in use, used to sequence entity cleanup in `drm_sched_entity_fini()`.

oldest_job_waiting

Marks earliest job waiting in SW queue

rb_tree_node

The node used to insert this entity into time based priority queue

Description

Entities will emit jobs in order to their corresponding hardware ring, and the scheduler will alternate between entities based on scheduling policy.

struct `drm_sched_rq`

queue of entities to be scheduled.

Definition:

```
struct drm_sched_rq {
    spinlock_t lock;
    struct drm_gpu_scheduler    *sched;
    struct list_head            entities;
    struct drm_sched_entity     *current_entity;
    struct rb_root_cached       rb_tree_root;
};
```

Members

lock

to modify the entities list.

sched

the scheduler to which this rq belongs to.

entities

list of the entities to be scheduled.

current_entity

the entity which is to be scheduled.

rb_tree_root

root of time based priority queue of entities for FIFO scheduling

Description

Run queue is a set of entities scheduling command submissions for one specific ring. It implements the scheduling policy that selects the next entity to emit commands from.

struct **drm_sched_fence**

fences corresponding to the scheduling of a job.

Definition:

```
struct drm_sched_fence {
    struct dma_fence    scheduled;
    struct dma_fence    finished;
    ktime_t deadline;
    struct dma_fence    *parent;
    struct drm_gpu_scheduler *sched;
    spinlock_t lock;
    void *owner;
};
```

Members

scheduled

this fence is what will be signaled by the scheduler when the job is scheduled.

finished

this fence is what will be signaled by the scheduler when the job is completed.

When setting up an out fence for the job, you should use this, since it's available immediately upon [drm_sched_job_init\(\)](#), and the fence returned by the driver from [run_job\(\)](#) won't be created until the dependencies have resolved.

deadline

deadline set on *drm_sched_fence.finished* which potentially needs to be propagated to *drm_sched_fence.parent*

parent

the fence returned by *drm_sched_backend_ops.run_job* when scheduling the job on hardware. We signal the *drm_sched_fence.finished* fence once parent is signalled.

sched

the scheduler instance to which the job having this struct belongs to.

lock

the lock used by the scheduled and the finished fences.

owner

job owner for debugging

struct *drm_sched_job*

A job to be run by an entity.

Definition:

```
struct drm_sched_job {
    struct spsc_node          queue_node;
    struct list_head          list;
    struct drm_gpu_scheduler  *sched;
    struct drm_sched_fence    *s_fence;
    union {
        struct dma_fence_cb    finish_cb;
        struct work_struct      work;
    };
    uint64_t id;
    atomic_t karma;
    enum drm_sched_priority    s_priority;
    struct drm_sched_entity    *entity;
    struct dma_fence_cb        cb;
    struct xarray              dependencies;
    unsigned long              last_dependency;
    ktime_t submit_ts;
};
```

Members**queue_node**

used to append this struct to the queue of jobs in an entity.

list

a job participates in a “pending” and “done” lists.

sched

the scheduler instance on which this job is scheduled.

s_fence

contains the fences for the scheduling of job.

{unnamed_union}

anonymous

finish_cb

the callback for the finished fence.

work

Helper to reschedule job kill to different context.

id

a unique id assigned to each job scheduled on the scheduler.

karma

increment on every hang caused by this job. If this exceeds the hang limit of the scheduler then the job is marked guilty and will not be scheduled further.

s_priority

the priority of the job.

entity

the entity to which this job belongs.

cb

the callback for the parent fence in s_fence.

dependencies

Contains the dependencies as struct dma_fence for this job, see [drm_sched_job_add_dependency\(\)](#) and [drm_sched_job_add_implicit_dependencies\(\)](#).

last_dependency

tracks **dependencies** as they signal

submit_ts

When the job was pushed into the entity queue.

Description

A job is created by the driver using [drm_sched_job_init\(\)](#), and should call [drm_sched_entity_push_job\(\)](#) once it wants the scheduler to schedule the job.

struct **drm_sched_backend_ops**

Define the backend operations called by the scheduler

Definition:

```
struct drm_sched_backend_ops {
    struct dma_fence *(*prepare_job)(struct drm_sched_job *sched_job, struct
→drm_sched_entity *s_entity);
    struct dma_fence *(*run_job)(struct drm_sched_job *sched_job);
    enum drm_gpu_sched_stat (*timedout_job)(struct drm_sched_job *sched_job);
    void (*free_job)(struct drm_sched_job *sched_job);
};
```

Members

prepare_job

Called when the scheduler is considering scheduling this job next, to get another struct dma_fence for this job to block on. Once it returns NULL, run_job() may be called.

Can be NULL if no additional preparation to the dependencies are necessary. Skipped when jobs are killed instead of run.

run_job

Called to execute the job once all of the dependencies have been resolved. This may be called multiple times, if `timedout_job()` has happened and `drm_sched_job_recovery()` decides to try it again.

timedout_job

Called when a job has taken too long to execute, to trigger GPU recovery.

This method is called in a workqueue context.

Drivers typically issue a reset to recover from GPU hangs, and this procedure usually follows the following workflow:

1. Stop the scheduler using `drm_sched_stop()`. This will park the scheduler thread and cancel the timeout work, guaranteeing that nothing is queued while we reset the hardware queue
2. Try to gracefully stop non-faulty jobs (optional)
3. Issue a GPU reset (driver-specific)
4. Re-submit jobs using `drm_sched_resubmit_jobs()`
5. Restart the scheduler using `drm_sched_start()`. At that point, new jobs can be queued, and the scheduler thread is unblocked

Note that some GPUs have distinct hardware queues but need to reset the GPU globally, which requires extra synchronization between the timeout handler of the different `drm_gpu_scheduler`. One way to achieve this synchronization is to create an ordered workqueue (using `alloc_ordered_workqueue()`) at the driver level, and pass this queue to `drm_sched_init()`, to guarantee that timeout handlers are executed sequentially. The above workflow needs to be slightly adjusted in that case:

1. Stop all schedulers impacted by the reset using `drm_sched_stop()`
2. Try to gracefully stop non-faulty jobs on all queues impacted by the reset (optional)
3. Issue a GPU reset on all faulty queues (driver-specific)
4. Re-submit jobs on all schedulers impacted by the reset using `drm_sched_resubmit_jobs()`
5. Restart all schedulers that were stopped in step #1 using `drm_sched_start()`

Return `DRM_GPU_SCHED_STAT_NOMINAL`, when all is normal, and the underlying driver has started or completed recovery.

Return `DRM_GPU_SCHED_STAT_ENODEV`, if the device is no longer available, i.e. has been unplugged.

free_job

Called once the job's finished fence has been signaled and it's time to clean it up.

Description

These functions should be implemented in the driver side.

struct `drm_gpu_scheduler`

scheduler instance-specific data

Definition:

```
struct drm_gpu_scheduler {
    const struct drm_sched_backend_ops      *ops;
    uint32_t hw_submission_limit;
    long timeout;
    const char                             *name;
    struct drm_sched_rq                    sched_rq[DRM_SCHED_PRIORITY_COUNT];
    wait_queue_head_t wake_up_worker;
    wait_queue_head_t job_scheduled;
    atomic_t hw_rq_count;
    atomic64_t job_id_count;
    struct workqueue_struct                *timeout_wq;
    struct delayed_work                    work_tdr;
    struct task_struct                     *thread;
    struct list_head                       pending_list;
    spinlock_t job_list_lock;
    int hang_limit;
    atomic_t *score;
    atomic_t _score;
    bool ready;
    bool free_guilty;
    struct device                          *dev;
};
```

Members

- ops**
backend operations provided by the driver.
- hw_submission_limit**
the max size of the hardware queue.
- timeout**
the time after which a job is removed from the scheduler.
- name**
name of the ring for which this scheduler is being used.
- sched_rq**
priority wise array of run queues.
- wake_up_worker**
the wait queue on which the scheduler sleeps until a job is ready to be scheduled.
- job_scheduled**
once **drm_sched_entity_do_release** is called the scheduler waits on this wait queue until all the scheduled jobs are finished.
- hw_rq_count**
the number of jobs currently in the hardware queue.
- job_id_count**
used to assign unique id to the each job.
- timeout_wq**
workqueue used to queue **work_tdr**

work_tdr

schedules a delayed call to **drm_sched_job_timeout** after the timeout interval is over.

thread

the kthread on which the scheduler which run.

pending_list

the list of jobs which are currently in the job queue.

job_list_lock

lock to protect the pending_list.

hang_limit

once the hangs by a job crosses this limit then it is marked guilty and it will no longer be considered for scheduling.

score

score to help loadbalancer pick a idle sched

_score

score used when the driver doesn't provide one

ready

marks if the underlying HW is ready to work

free_guilty

A hit to time out handler to free the guilty job.

dev

system struct device

Description

One scheduler is implemented for each hardware ring.

void **drm_sched_fault**(struct *drm_gpu_scheduler* *sched)

immediately start timeout handler

Parameters

struct drm_gpu_scheduler *sched

scheduler where the timeout handling should be started.

Description

Start timeout handling immediately when the driver detects a hardware fault.

unsigned long **drm_sched_suspend_timeout**(struct *drm_gpu_scheduler* *sched)

Suspend scheduler job timeout

Parameters

struct drm_gpu_scheduler *sched

scheduler instance for which to suspend the timeout

Description

Suspend the delayed work timeout for the scheduler. This is done by modifying the delayed work timeout to an arbitrary large value, MAX_SCHEDULE_TIMEOUT in this case.

Returns the timeout remaining

void **drm_sched_resume_timeout**(struct *drm_gpu_scheduler* *sched, unsigned long remaining)

Resume scheduler job timeout

Parameters

struct drm_gpu_scheduler *sched
scheduler instance for which to resume the timeout

unsigned long remaining
remaining timeout

Description

Resume the delayed work timeout for the scheduler.

void **drm_sched_stop**(struct *drm_gpu_scheduler* *sched, struct *drm_sched_job* *bad)
stop the scheduler

Parameters

struct drm_gpu_scheduler *sched
scheduler instance

struct drm_sched_job *bad
job which caused the time out

Description

Stop the scheduler and also removes and frees all completed jobs.

Note

bad job will not be freed as it might be used later and so it's callers responsibility to release it manually if it's not part of the pending list any more.

void **drm_sched_start**(struct *drm_gpu_scheduler* *sched, bool full_recovery)
recover jobs after a reset

Parameters

struct drm_gpu_scheduler *sched
scheduler instance

bool full_recovery
proceed with complete sched restart

void **drm_sched_resubmit_jobs**(struct *drm_gpu_scheduler* *sched)
Deprecated, don't use in new code!

Parameters

struct drm_gpu_scheduler *sched
scheduler instance

Description

Re-submitting jobs was a concept AMD came up as cheap way to implement recovery after a job timeout.

This turned out to be not working very well. First of all there are many problem with the `dma_fence` implementation and requirements. Either the implementation is risking deadlocks with core memory management or violating documented implementation details of the `dma_fence` object.

Drivers can still save and restore their state for recovery operations, but we shouldn't make this a general scheduler feature around the `dma_fence` interface.

```
int drm_sched_job_init(struct drm_sched_job *job, struct drm_sched_entity *entity, void
                      *owner)
```

init a scheduler job

Parameters

```
struct drm_sched_job *job
```

scheduler job to init

```
struct drm_sched_entity *entity
```

scheduler entity to use

```
void *owner
```

job owner for debugging

Description

Refer to `drm_sched_entity_push_job()` documentation for locking considerations.

Drivers must make sure `drm_sched_job_cleanup()` if this function returns successfully, even when **job** is aborted before `drm_sched_job_arm()` is called.

WARNING: amdgpu abuses `drm_sched.ready` to signal when the hardware has died, which can mean that there's no valid runqueue for a **entity**. This function returns `-ENOENT` in this case (which probably should be `-EIO` as a more meaningful return value).

Returns 0 for success, negative error code otherwise.

```
void drm_sched_job_arm(struct drm_sched_job *job)
```

arm a scheduler job for execution

Parameters

```
struct drm_sched_job *job
```

scheduler job to arm

Description

This arms a scheduler job for execution. Specifically it initializes the `drm_sched_job.s_fence` of **job**, so that it can be attached to struct `dma_resv` or other places that need to track the completion of this job.

Refer to `drm_sched_entity_push_job()` documentation for locking considerations.

This can only be called if `drm_sched_job_init()` succeeded.

```
int drm_sched_job_add_dependency(struct drm_sched_job *job, struct dma_fence *fence)
```

adds the fence as a job dependency

Parameters

```
struct drm_sched_job *job
```

scheduler job to add the dependencies to

struct dma_fence *fence

the dma_fence to add to the list of dependencies.

Description

Note that **fence** is consumed in both the success and error cases.

Return

0 on success, or an error on failing to expand the array.

int **drm_sched_job_add_syncobj_dependency**(struct *drm_sched_job* *job, struct *drm_file* *file,
u32 handle, u32 point)

adds a syncobj's fence as a job dependency

Parameters

struct drm_sched_job *job

scheduler job to add the dependencies to

struct drm_file *file

drm file private pointer

u32 handle

syncobj handle to lookup

u32 point

timeline point

Description

This adds the fence matching the given syncobj to **job**.

Return

0 on success, or an error on failing to expand the array.

int **drm_sched_job_add_resv_dependencies**(struct *drm_sched_job* *job, struct dma_resv
*resv, enum dma_resv_usage usage)

add all fences from the resv to the job

Parameters

struct drm_sched_job *job

scheduler job to add the dependencies to

struct dma_resv *resv

the dma_resv object to get the fences from

enum dma_resv_usage usage

the dma_resv_usage to use to filter the fences

Description

This adds all fences matching the given usage from **resv** to **job**. Must be called with the **resv** lock held.

Return

0 on success, or an error on failing to expand the array.

int **drm_sched_job_add_implicit_dependencies**(struct *drm_sched_job* *job, struct *drm_gem_object* *obj, bool write)

adds implicit dependencies as job dependencies

Parameters

struct *drm_sched_job* *job

scheduler job to add the dependencies to

struct *drm_gem_object* *obj

the gem object to add new dependencies from.

bool write

whether the job might write the object (so we need to depend on shared fences in the reservation object).

Description

This should be called after *drm_gem_lock_reservations()* on your array of GEM objects used in the job but before updating the reservations with your own fences.

Return

0 on success, or an error on failing to expand the array.

void **drm_sched_job_cleanup**(struct *drm_sched_job* *job)

clean up scheduler job resources

Parameters

struct *drm_sched_job* *job

scheduler job to clean up

Description

Cleans up the resources allocated with *drm_sched_job_init()*.

Drivers should call this from their error unwind code if **job** is aborted before *drm_sched_job_arm()* is called.

After that point of no return **job** is committed to be executed by the scheduler, and this function should be called from the *drm_sched_backend_ops.free_job* callback.

struct *drm_gpu_scheduler* ***drm_sched_pick_best**(struct *drm_gpu_scheduler* **sched_list, unsigned int num_sched_list)

Get a drm sched from a sched_list with the least load

Parameters

struct *drm_gpu_scheduler* **sched_list

list of *drm_gpu_schedulers*

unsigned int num_sched_list

number of *drm_gpu_schedulers* in the sched_list

Description

Returns pointer of the sched with the least load or NULL if none of the *drm_gpu_schedulers* are ready

int **drm_sched_init**(struct *drm_gpu_scheduler* *sched, const struct *drm_sched_backend_ops* *ops, unsigned hw_submission, unsigned hang_limit, long timeout, struct workqueue_struct *timeout_wq, atomic_t *score, const char *name, struct device *dev)

Init a gpu scheduler instance

Parameters

struct drm_gpu_scheduler *sched
scheduler instance

const struct drm_sched_backend_ops *ops
backend operations for this scheduler

unsigned hw_submission
number of hw submissions that can be in flight

unsigned hang_limit
number of times to allow a job to hang before dropping it

long timeout
timeout value in jiffies for the scheduler

struct workqueue_struct *timeout_wq
workqueue to use for timeout work. If NULL, the system_wq is used

atomic_t *score
optional score atomic shared with other schedulers

const char *name
name used for debugging

struct device *dev
target struct device

Description

Return 0 on success, otherwise error code.

void **drm_sched_fini**(struct *drm_gpu_scheduler* *sched)
Destroy a gpu scheduler

Parameters

struct drm_gpu_scheduler *sched
scheduler instance

Description

Tears down and cleans up the scheduler.

void **drm_sched_increase_karma**(struct *drm_sched_job* *bad)
Update sched_entity guilty flag

Parameters

struct drm_sched_job *bad
The job guilty of time out

Description

Increment on every hang caused by the 'bad' job. If this exceeds the hang limit of the scheduler then the respective sched entity is marked guilty and jobs from it will not be scheduled further

```
int drm_sched_entity_init(struct drm_sched_entity *entity, enum drm_sched_priority
                        priority, struct drm_gpu_scheduler **sched_list, unsigned int
                        num_sched_list, atomic_t *guilty)
```

Init a context entity used by scheduler when submit to HW ring.

Parameters

struct *drm_sched_entity* *entity
scheduler entity to init

enum *drm_sched_priority* priority
priority of the entity

struct *drm_gpu_scheduler* **sched_list
the list of drm scheds on which jobs from this entity can be submitted

unsigned int num_sched_list
number of drm sched in sched_list

atomic_t *guilty
atomic_t set to 1 when a job on this queue is found to be guilty causing a timeout

Description

Note that the sched_list must have at least one element to schedule the entity.

For changing **priority** later on at runtime see *drm_sched_entity_set_priority()*. For changing the set of schedulers **sched_list** at runtime see *drm_sched_entity_modify_sched()*.

An entity is cleaned up by calling *drm_sched_entity_fini()*. See also *drm_sched_entity_destroy()*.

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

```
void drm_sched_entity_modify_sched(struct drm_sched_entity *entity, struct
                                drm_gpu_scheduler **sched_list, unsigned int
                                num_sched_list)
```

Modify sched of an entity

Parameters

struct *drm_sched_entity* *entity
scheduler entity to init

struct *drm_gpu_scheduler* **sched_list
the list of new drm scheds which will replace existing entity->sched_list

unsigned int num_sched_list
number of drm sched in sched_list

Description

Note that this must be called under the same common lock for **entity** as *drm_sched_job_arm()* and *drm_sched_entity_push_job()*, or the driver needs to guarantee through some other means that this is never called while new jobs can be pushed to **entity**.

int **drm_sched_entity_error**(struct *drm_sched_entity* *entity)
return error of last scheduled job

Parameters

struct drm_sched_entity *entity
scheduler entity to check

Description

Opportunistically return the error of the last scheduled job. Result can change any time when new jobs are pushed to the hw.

long **drm_sched_entity_flush**(struct *drm_sched_entity* *entity, long timeout)
Flush a context entity

Parameters

struct drm_sched_entity *entity
scheduler entity

long timeout
time to wait in for Q to become empty in jiffies.

Description

Splitting *drm_sched_entity_fini()* into two functions, The first one does the waiting, removes the entity from the runqueue and returns an error when the process was killed.

Returns the remaining time in jiffies left from the input timeout

void **drm_sched_entity_fini**(struct *drm_sched_entity* *entity)
Destroy a context entity

Parameters

struct drm_sched_entity *entity
scheduler entity

Description

Cleanups up **entity** which has been initialized by *drm_sched_entity_init()*.

If there are potentially job still in flight or getting newly queued *drm_sched_entity_flush()* must be called first. This function then goes over the entity and signals all jobs with an error code if the process was killed.

void **drm_sched_entity_destroy**(struct *drm_sched_entity* *entity)
Destroy a context entity

Parameters

struct drm_sched_entity *entity
scheduler entity

Description

Calls *drm_sched_entity_flush()* and *drm_sched_entity_fini()* as a convenience wrapper.

void **drm_sched_entity_set_priority**(struct *drm_sched_entity* *entity, enum
drm_sched_priority priority)
Sets priority of the entity

Parameters

struct `drm_sched_entity *entity`
scheduler entity

enum `drm_sched_priority priority`
scheduler priority

Description

Update the priority of runqueue used for the entity.

void `drm_sched_entity_push_job`(**struct** *drm_sched_job* *`sched_job`)
Submit a job to the entity's job queue

Parameters

struct `drm_sched_job *sched_job`
job to submit

Note

To guarantee that the order of insertion to queue matches the job's fence sequence number this function should be called with *drm_sched_job_arm()* under common lock for the *struct drm_sched_entity* that was set up for **sched_job** in *drm_sched_job_init()*.

Description

Returns 0 for success, negative error code otherwise.

KERNEL MODE SETTING (KMS)

Drivers must initialize the mode setting core by calling `drm_mode_config_init()` on the DRM device. The function initializes the `struct 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.

4.1 Overview

The basic object structure KMS presents to userspace is fairly simple. Framebuffers (represented by `struct drm_framebuffer`, see *Frame Buffer Abstraction*) feed into planes. Planes are represented by `struct drm_plane`, see *Plane Abstraction* for more details. One or more (or even no) planes feed their pixel data into a CRTC (represented by `struct drm_crtc`, see *CRTC Abstraction*) for blending. The precise blending step is explained in more detail in *Plane Composition Properties* and related chapters.

For the output routing the first step is encoders (represented by `struct drm_encoder`, see *Encoder Abstraction*). Those are really just internal artifacts of the helper libraries used to implement KMS drivers. Besides that they make it unnecessarily more complicated for userspace to figure out which connections between a CRTC and a connector are possible, and what kind of cloning is supported, they serve no purpose in the userspace API. Unfortunately encoders have been exposed to userspace, hence can't remove them at this point. Furthermore the exposed restrictions are often wrongly set by drivers, and in many cases not powerful enough to express the real restrictions. A CRTC can be connected to multiple encoders, and for an active CRTC there must be at least one encoder.

The final, and real, endpoint in the display chain is the connector (represented by `struct drm_connector`, see *Connector Abstraction*). Connectors can have different possible encoders, but the kernel driver selects which encoder to use for each connector. The use case is DVI, which could switch between an analog and a digital encoder. Encoders can also drive multiple different connectors. There is exactly one active connector for every active encoder.

Internally the output pipeline is a bit more complex and matches today's hardware more closely:

Internally two additional helper objects come into play. First, to be able to share code for encoders (sometimes on the same SoC, sometimes off-chip) one or more *Bridges* (represented by `struct drm_bridge`) can be linked to an encoder. This link is static and cannot be changed, which means the cross-bar (if there is any) needs to be mapped between the CRTC and any encoders. Often for drivers with bridges there's no code left at the encoder level. Atomic

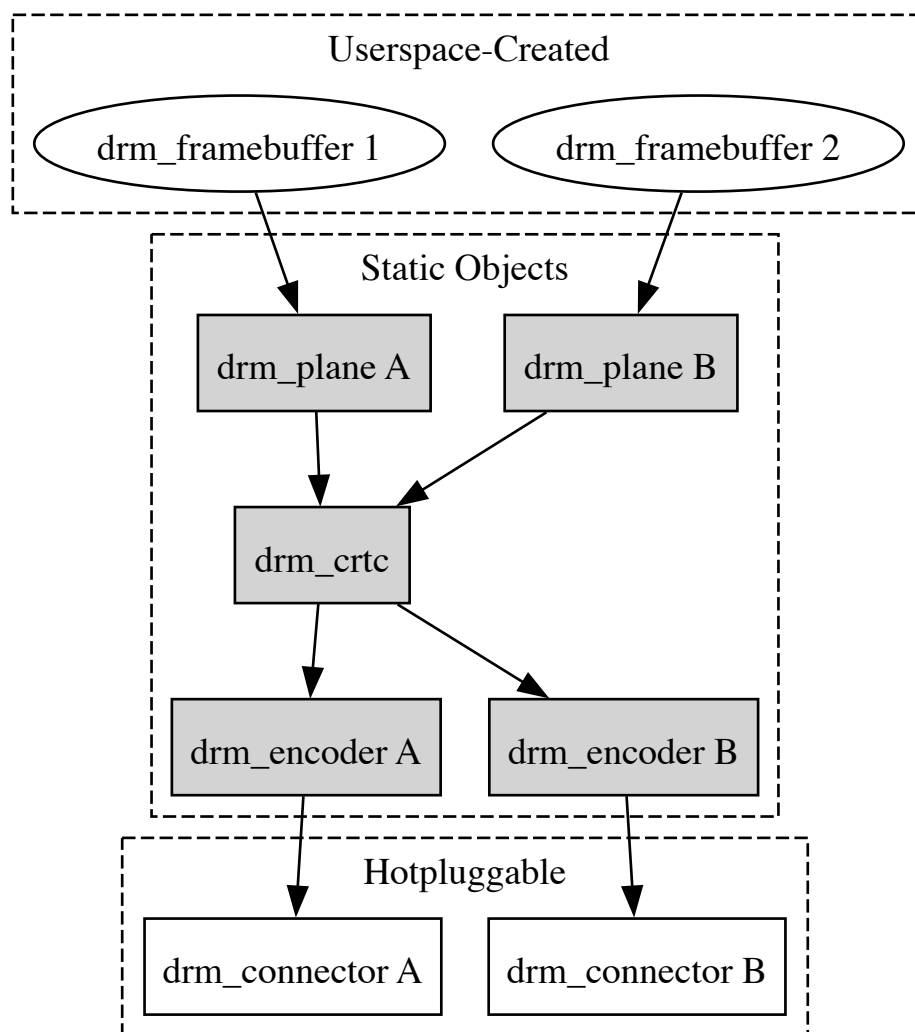


Fig. 1: KMS Display Pipeline Overview

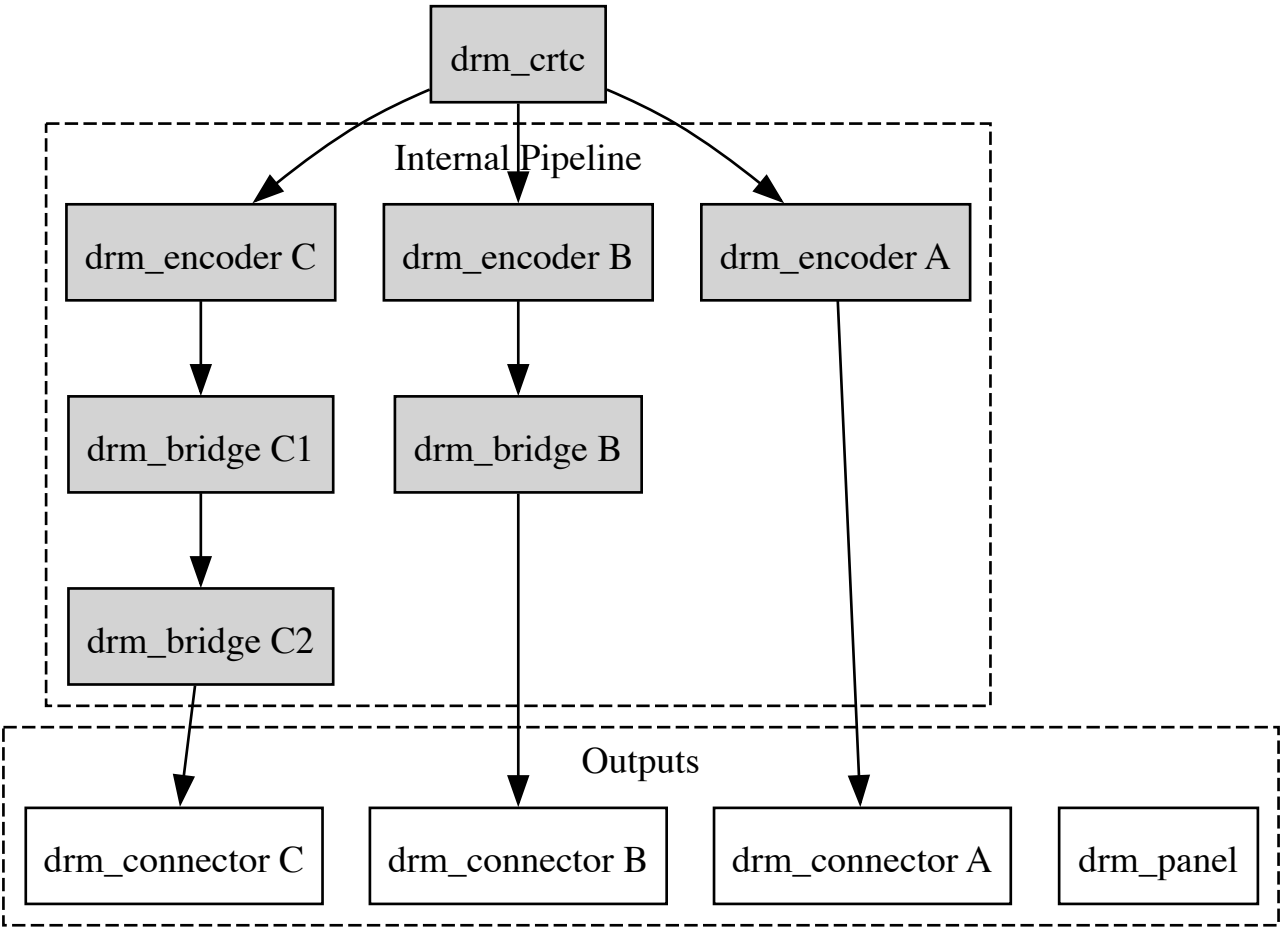


Fig. 2: KMS Output Pipeline

drivers can leave out all the encoder callbacks to essentially only leave a dummy routing object behind, which is needed for backwards compatibility since encoders are exposed to userspace.

The second object is for panels, represented by *struct [drm_panel](#)*, see *Panel Helper Reference*. Panels do not have a fixed binding point, but are generally linked to the driver private structure that embeds *struct [drm_connector](#)*.

Note that currently the bridge chaining and interactions with connectors and panels are still in-flux and not really fully sorted out yet.

4.2 KMS Core Structures and Functions

struct **drm_mode_config_funcs**

basic driver provided mode setting functions

Definition:

```
struct drm_mode_config_funcs {
    struct drm_framebuffer *(*fb_create)(struct drm_device *dev, struct drm_
↪ file *file_priv, const struct drm_mode_fb_cmd2 *mode_cmd);
    const struct drm_format_info *(*get_format_info)(const struct drm_mode_fb_
↪ cmd2 *mode_cmd);
    void (*output_poll_changed)(struct drm_device *dev);
    enum drm_mode_status (*mode_valid)(struct drm_device *dev, const struct_
↪ drm_display_mode *mode);
    int (*atomic_check)(struct drm_device *dev, struct drm_atomic_state_
↪ *state);
    int (*atomic_commit)(struct drm_device *dev, struct drm_atomic_state *state,
↪ bool nonblock);
    struct drm_atomic_state *(*atomic_state_alloc)(struct drm_device *dev);
    void (*atomic_state_clear)(struct drm_atomic_state *state);
    void (*atomic_state_free)(struct drm_atomic_state *state);
};
```

Members

fb_create

Create a new framebuffer object. The core does basic checks on the requested metadata, but most of that is left to the driver. See *struct [drm_mode_fb_cmd2](#)* for details.

To validate the pixel format and modifier drivers can use *[drm_any_plane_has_format\(\)](#)* to make sure at least one plane supports the requested values. Note that the driver must first determine the actual modifier used if the request doesn't have it specified, ie. when **(mode_cmd->flags & DRM_MODE_FB_MODIFIERS) == 0**.

IMPORTANT: These implied modifiers for legacy userspace must be stored in struct *[drm_framebuffer](#)*, including all relevant metadata like *[drm_framebuffer.pitches](#)* and *[drm_framebuffer.offsets](#)* if the modifier enables additional planes beyond the fourcc pixel format code. This is required by the GETFB2 ioctl.

If the parameters are deemed valid and the backing storage objects in the underlying memory manager all exist, then the driver allocates a new *[drm_framebuffer](#)* structure,

subclassed to contain driver-specific information (like the internal native buffer object references). It also needs to fill out all relevant metadata, which should be done by calling `drm_helper_mode_fill_fb_struct()`.

The initialization is finalized by calling `drm_framebuffer_init()`, which registers the framebuffer and makes it accessible to other threads.

RETURNS:

A new framebuffer with an initial reference count of 1 or a negative error code encoded with `ERR_PTR()`.

get_format_info

Allows a driver to return custom format information for special fb layouts (eg. ones with auxiliary compression control planes).

RETURNS:

The format information specific to the given fb metadata, or NULL if none is found.

output_poll_changed

Callback used by helpers to inform the driver of output configuration changes.

Drivers implementing fbdev emulation use `drm_kms_helper_hotplug_event()` to call this hook to inform the fbdev helper of output changes.

This hook is deprecated, drivers should instead use `drm_fbdev_generic_setup()` which takes care of any necessary hotplug event forwarding already without further involvement by the driver.

mode_valid

Device specific validation of display modes. Can be used to reject modes that can never be supported. Only device wide constraints can be checked here. crtc/encoder/bridge/connector specific constraints should be checked in the `.mode_valid()` hook for each specific object.

atomic_check

This is the only hook to validate an atomic modeset update. This function must reject any modeset and state changes which the hardware or driver doesn't support. This includes but is of course not limited to:

- Checking that the modes, framebuffers, scaling and placement requirements and so on are within the limits of the hardware.
- Checking that any hidden shared resources are not oversubscribed. This can be shared PLLs, shared lanes, overall memory bandwidth, display fifo space (where shared between planes or maybe even CRTCs).
- Checking that virtualized resources exported to userspace are not oversubscribed. For various reasons it can make sense to expose more planes, crtcs or encoders than which are physically there. One example is dual-pipe operations (which generally should be hidden from userspace if when lockstepped in hardware, exposed otherwise), where a plane might need 1 hardware plane (if it's just on one pipe), 2 hardware planes (when it spans both pipes) or maybe even shared a hardware plane with a 2nd plane (if there's a compatible plane requested on the area handled by the other pipe).
- Check that any transitional state is possible and that if requested, the update can indeed be done in the vblank period without temporarily disabling some functions.

- Check any other constraints the driver or hardware might have.
- This callback also needs to correctly fill out the `drm_crtc_state` in this update to make sure that `drm_atomic_crtc_needs_modeset()` reflects the nature of the possible update and returns true if and only if the update cannot be applied without tearing within one vblank on that CRTC. The core uses that information to reject updates which require a full modeset (i.e. blanking the screen, or at least pausing updates for a substantial amount of time) if userspace has disallowed that in its request.
- The driver also does not need to repeat basic input validation like done for the corresponding legacy entry points. The core does that before calling this hook.

See the documentation of **atomic_commit** for an exhaustive list of error conditions which don't have to be checked at the in this callback.

See the documentation for `struct drm_atomic_state` for how exactly an atomic modeset update is described.

Drivers using the atomic helpers can implement this hook using `drm_atomic_helper_check()`, or one of the exported sub-functions of it.

RETURNS:

0 on success or one of the below negative error codes:

- -EINVAL, if any of the above constraints are violated.
- -EDEADLK, when returned from an attempt to acquire an additional `drm_modeset_lock` through `drm_modeset_lock()`.
- -ENOMEM, if allocating additional state sub-structures failed due to lack of memory.
- -EINTR, -EAGAIN or -ERESTARTSYS, if the IOCTL should be restarted. This can either be due to a pending signal, or because the driver needs to completely bail out to recover from an exceptional situation like a GPU hang. From a userspace point all errors are treated equally.

atomic_commit

This is the only hook to commit an atomic modeset update. The core guarantees that **atomic_check** has been called successfully before calling this function, and that nothing has been changed in the interim.

See the documentation for `struct drm_atomic_state` for how exactly an atomic modeset update is described.

Drivers using the atomic helpers can implement this hook using `drm_atomic_helper_commit()`, or one of the exported sub-functions of it.

Nonblocking commits (as indicated with the `nonblock` parameter) must do any preparatory work which might result in an unsuccessful commit in the context of this callback. The only exceptions are hardware errors resulting in -EIO. But even in that case the driver must ensure that the display pipe is at least running, to avoid compositors crashing when pageflips don't work. Anything else, specifically committing the update to the hardware, should be done without blocking the caller. For updates which do not require a modeset this must be guaranteed.

The driver must wait for any pending rendering to the new framebuffers to complete before executing the flip. It should also wait for any pending rendering from other drivers if the

underlying buffer is a shared dma-buf. Nonblocking commits must not wait for rendering in the context of this callback.

An application can request to be notified when the atomic commit has completed. These events are per-CRTC and can be distinguished by the CRTC index supplied in *drm_event* to userspace.

The drm core will supply a *struct drm_event* in each CRTC's *drm_crtc_state.event*. See the documentation for *drm_crtc_state.event* for more details about the precise semantics of this event.

NOTE:

Drivers are not allowed to shut down any display pipe successfully enabled through an atomic commit on their own. Doing so can result in compositors crashing if a page flip is suddenly rejected because the pipe is off.

RETURNS:

0 on success or one of the below negative error codes:

- -EBUSY, if a nonblocking update is requested and there is an earlier update pending. Drivers are allowed to support a queue of outstanding updates, but currently no driver supports that. Note that drivers must wait for preceding updates to complete if a synchronous update is requested, they are not allowed to fail the commit in that case.
- -ENOMEM, if the driver failed to allocate memory. Specifically this can happen when trying to pin framebuffers, which must only be done when committing the state.
- -ENOSPC, as a refinement of the more generic -ENOMEM to indicate that the driver has run out of vram, iommu space or similar GPU address space needed for framebuffer.
- -EIO, if the hardware completely died.
- -EINTR, -EAGAIN or -ERESTARTSYS, if the IOCTL should be restarted. This can either be due to a pending signal, or because the driver needs to completely bail out to recover from an exceptional situation like a GPU hang. From a userspace point of view all errors are treated equally.

This list is exhaustive. Specifically this hook is not allowed to return -EINVAL (any invalid requests should be caught in **atomic_check**) or -EDEADLK (this function must not acquire additional modeset locks).

atomic_state_alloc

This optional hook can be used by drivers that want to subclass *struct drm_atomic_state* to be able to track their own driver-private global state easily. If this hook is implemented, drivers must also implement **atomic_state_clear** and **atomic_state_free**.

Subclassing of *drm_atomic_state* is deprecated in favour of using *drm_private_state* and *drm_private_obj*.

RETURNS:

A new *drm_atomic_state* on success or NULL on failure.

atomic_state_clear

This hook must clear any driver private state duplicated into the passed-in *drm_atomic_state*. This hook is called when the caller encountered a *drm_modeset_lock*

deadlock and needs to drop all already acquired locks as part of the deadlock avoidance dance implemented in `drm_modeset_backoff()`.

Any duplicated state must be invalidated since a concurrent atomic update might change it, and the drm atomic interfaces always apply updates as relative changes to the current state.

Drivers that implement this must call `drm_atomic_state_default_clear()` to clear common state.

Subclassing of `drm_atomic_state` is deprecated in favour of using `drm_private_state` and `drm_private_obj`.

atomic_state_free

This hook needs driver private resources and the `drm_atomic_state` itself. Note that the core first calls `drm_atomic_state_clear()` to avoid code duplicate between the clear and free hooks.

Drivers that implement this must call `drm_atomic_state_default_release()` to release common resources.

Subclassing of `drm_atomic_state` is deprecated in favour of using `drm_private_state` and `drm_private_obj`.

Description

Some global (i.e. not per-CRTC, connector, etc) mode setting functions that involve drivers.

struct drm_mode_config

Mode configuration control structure

Definition:

```
struct drm_mode_config {
    struct mutex mutex;
    struct drm_modeset_lock connection_mutex;
    struct drm_modeset_acquire_ctx *acquire_ctx;
    struct mutex idr_mutex;
    struct idr object_idr;
    struct idr tile_idr;
    struct mutex fb_lock;
    int num_fb;
    struct list_head fb_list;
    spinlock_t connector_list_lock;
    int num_connector;
    struct ida connector_ida;
    struct list_head connector_list;
    struct llist_head connector_free_list;
    struct work_struct connector_free_work;
    int num_encoder;
    struct list_head encoder_list;
    int num_total_plane;
    struct list_head plane_list;
    int num_crtc;
    struct list_head crtc_list;
    struct list_head property_list;
```

```

struct list_head privobj_list;
int min_width, min_height;
int max_width, max_height;
const struct drm_mode_config_funcs *funcs;
bool poll_enabled;
bool poll_running;
bool delayed_event;
struct delayed_work output_poll_work;
struct mutex blob_lock;
struct list_head property_blob_list;
struct drm_property *edid_property;
struct drm_property *dpms_property;
struct drm_property *path_property;
struct drm_property *tile_property;
struct drm_property *link_status_property;
struct drm_property *plane_type_property;
struct drm_property *prop_src_x;
struct drm_property *prop_src_y;
struct drm_property *prop_src_w;
struct drm_property *prop_src_h;
struct drm_property *prop_crtc_x;
struct drm_property *prop_crtc_y;
struct drm_property *prop_crtc_w;
struct drm_property *prop_crtc_h;
struct drm_property *prop_fb_id;
struct drm_property *prop_in_fence_fd;
struct drm_property *prop_out_fence_ptr;
struct drm_property *prop_crtc_id;
struct drm_property *prop_fb_damage_clips;
struct drm_property *prop_active;
struct drm_property *prop_mode_id;
struct drm_property *prop_vrr_enabled;
struct drm_property *dvi_i_subconnector_property;
struct drm_property *dvi_i_select_subconnector_property;
struct drm_property *dp_subconnector_property;
struct drm_property *tv_subconnector_property;
struct drm_property *tv_select_subconnector_property;
struct drm_property *legacy_tv_mode_property;
struct drm_property *tv_mode_property;
struct drm_property *tv_left_margin_property;
struct drm_property *tv_right_margin_property;
struct drm_property *tv_top_margin_property;
struct drm_property *tv_bottom_margin_property;
struct drm_property *tv_brightness_property;
struct drm_property *tv_contrast_property;
struct drm_property *tv_flicker_reduction_property;
struct drm_property *tv_overscan_property;
struct drm_property *tv_saturation_property;
struct drm_property *tv_hue_property;
struct drm_property *scaling_mode_property;

```

```
struct drm_property *aspect_ratio_property;
struct drm_property *content_type_property;
struct drm_property *degamma_lut_property;
struct drm_property *degamma_lut_size_property;
struct drm_property *ctm_property;
struct drm_property *gamma_lut_property;
struct drm_property *gamma_lut_size_property;
struct drm_property *suggested_x_property;
struct drm_property *suggested_y_property;
struct drm_property *non_desktop_property;
struct drm_property *panel_orientation_property;
struct drm_property *writeback_fb_id_property;
struct drm_property *writeback_pixel_formats_property;
struct drm_property *writeback_out_fence_ptr_property;
struct drm_property *hdr_output_metadata_property;
struct drm_property *content_protection_property;
struct drm_property *hdc_content_type_property;
uint32_t preferred_depth, prefer_shadow;
bool quirk_addfb_prefer_xbgr_30bpp;
bool quirk_addfb_prefer_host_byte_order;
bool async_page_flip;
bool fb_modifiers_not_supported;
bool normalize_zpos;
struct drm_property *modifiers_property;
uint32_t cursor_width, cursor_height;
struct drm_atomic_state *suspend_state;
const struct drm_mode_config_helper_funcs *helper_private;
};
```

Members

mutex

This is the big scary modeset BKL which protects everything that isn't protect otherwise. Scope is unclear and fuzzy, try to remove anything from under its protection and move it into more well-scoped locks.

The one important thing this protects is the use of **acquire_ctx**.

connection_mutex

This protects connector state and the connector to encoder to CRTC routing chain.

For atomic drivers specifically this protects *drm_connector.state*.

acquire_ctx

Global implicit acquire context used by atomic drivers for legacy IOCTLs. Deprecated, since implicit locking contexts make it impossible to use driver-private *struct drm_modeset_lock*. Users of this must hold **mutex**.

idr_mutex

Mutex for KMS ID allocation and management. Protects both **object_idr** and **tile_idr**.

object_idr

Main KMS ID tracking object. Use this idr for all IDs, fb, crtc, connector, modes - just makes life easier to have only one.

tile_idr

Use this idr for allocating new IDs for tiled sinks like use in some high-res DP MST screens.

fb_lock

Mutex to protect fb the global **fb_list** and **num_fb**.

num_fb

Number of entries on **fb_list**.

fb_list

List of all *struct drm_framebuffer*.

connector_list_lock

Protects **num_connector** and **connector_list** and **connector_free_list**.

num_connector

Number of connectors on this device. Protected by **connector_list_lock**.

connector_ida

ID allocator for connector indices.

connector_list

List of connector objects linked with *drm_connector.head*. Protected by **connector_list_lock**. Only use *drm_for_each_connector_iter()* and *struct drm_connector_list_iter* to walk this list.

connector_free_list

List of connector objects linked with *drm_connector.free_head*. Protected by **connector_list_lock**. Used by *drm_for_each_connector_iter()* and *struct drm_connector_list_iter* to safely free connectors using **connector_free_work**.

connector_free_work

Work to clean up **connector_free_list**.

num_encoder

Number of encoders on this device. This is invariant over the lifetime of a device and hence doesn't need any locks.

encoder_list

List of encoder objects linked with *drm_encoder.head*. This is invariant over the lifetime of a device and hence doesn't need any locks.

num_total_plane

Number of universal (i.e. with primary/cursor) planes on this device. This is invariant over the lifetime of a device and hence doesn't need any locks.

plane_list

List of plane objects linked with *drm_plane.head*. This is invariant over the lifetime of a device and hence doesn't need any locks.

num_crtc

Number of CRTC on this device linked with *drm_crtc.head*. This is invariant over the lifetime of a device and hence doesn't need any locks.

crtc_list

List of CRTC objects linked with *drm_crtc.head*. This is invariant over the lifetime of a device and hence doesn't need any locks.

property_list

List of property type objects linked with *drm_property.head*. This is invariant over the lifetime of a device and hence doesn't need any locks.

privobj_list

List of private objects linked with *drm_private_obj.head*. This is invariant over the lifetime of a device and hence doesn't need any locks.

min_width

minimum fb pixel width on this device

min_height

minimum fb pixel height on this device

max_width

maximum fb pixel width on this device

max_height

maximum fb pixel height on this device

funcs

core driver provided mode setting functions

poll_enabled

track polling support for this device

poll_running

track polling status for this device

delayed_event

track delayed poll uevent deliver for this device

output_poll_work

delayed work for polling in process context

blob_lock

Mutex for blob property allocation and management, protects **property_blob_list** and *drm_file.blobs*.

property_blob_list

List of all the blob property objects linked with *drm_property_blob.head*. Protected by **blob_lock**.

edid_property

Default connector property to hold the EDID of the currently connected sink, if any.

dpms_property

Default connector property to control the connector's DPMS state.

path_property

Default connector property to hold the DP MST path for the port.

tile_property

Default connector property to store the tile position of a tiled screen, for sinks which need to be driven with multiple CRTCs.

link_status_property

Default connector property for link status of a connector

plane_type_property

Default plane property to differentiate CURSOR, PRIMARY and OVERLAY legacy uses of planes.

prop_src_x

Default atomic plane property for the plane source position in the connected *drm_framebuffer*.

prop_src_y

Default atomic plane property for the plane source position in the connected *drm_framebuffer*.

prop_src_w

Default atomic plane property for the plane source position in the connected *drm_framebuffer*.

prop_src_h

Default atomic plane property for the plane source position in the connected *drm_framebuffer*.

prop_crtc_x

Default atomic plane property for the plane destination position in the *drm_crtc* is being shown on.

prop_crtc_y

Default atomic plane property for the plane destination position in the *drm_crtc* is being shown on.

prop_crtc_w

Default atomic plane property for the plane destination position in the *drm_crtc* is being shown on.

prop_crtc_h

Default atomic plane property for the plane destination position in the *drm_crtc* is being shown on.

prop_fb_id

Default atomic plane property to specify the *drm_framebuffer*.

prop_in_fence_fd

Sync File fd representing the incoming fences for a Plane.

prop_out_fence_ptr

Sync File fd pointer representing the outgoing fences for a CRTC. Userspace should provide a pointer to a value of type s32, and then cast that pointer to u64.

prop_crtc_id

Default atomic plane property to specify the *drm_crtc*.

prop_fb_damage_clips

Optional plane property to mark damaged regions on the plane in framebuffer coordinates of the framebuffer attached to the plane.

The layout of blob data is simply an array of *drm_mode_rect*. Unlike plane src coordinates, damage clips are not in 16.16 fixed point.

prop_active

Default atomic CRTC property to control the active state, which is the simplified implementation for DPMS in atomic drivers.

prop_mode_id

Default atomic CRTC property to set the mode for a CRTC. A 0 mode implies that the CRTC is entirely disabled - all connectors must be of and active must be set to disabled, too.

prop_vrr_enabled

Default atomic CRTC property to indicate whether variable refresh rate should be enabled on the CRTC.

dvi_i_subconnector_property

Optional DVI-I property to differentiate between analog or digital mode.

dvi_i_select_subconnector_property

Optional DVI-I property to select between analog or digital mode.

dp_subconnector_property

Optional DP property to differentiate between different DP downstream port types.

tv_subconnector_property

Optional TV property to differentiate between different TV connector types.

tv_select_subconnector_property

Optional TV property to select between different TV connector types.

legacy_tv_mode_property

Optional TV property to select the output TV mode.

Superseded by **tv_mode_property**

tv_mode_property

Optional TV property to select the TV standard output on the connector.

tv_left_margin_property

Optional TV property to set the left margin (expressed in pixels).

tv_right_margin_property

Optional TV property to set the right margin (expressed in pixels).

tv_top_margin_property

Optional TV property to set the right margin (expressed in pixels).

tv_bottom_margin_property

Optional TV property to set the right margin (expressed in pixels).

tv_brightness_property

Optional TV property to set the brightness.

tv_contrast_property

Optional TV property to set the contrast.

tv_flicker_reduction_property

Optional TV property to control the flicker reduction mode.

tv_overscan_property

Optional TV property to control the overscan setting.

tv_saturation_property

Optional TV property to set the saturation.

tv_hue_property

Optional TV property to set the hue.

scaling_mode_property

Optional connector property to control the upscaling, mostly used for built-in panels.

aspect_ratio_property

Optional connector property to control the HDMI infoframe aspect ratio setting.

content_type_property

Optional connector property to control the HDMI infoframe content type setting.

degamma_lut_property

Optional CRTC property to set the LUT used to convert the framebuffer's colors to linear gamma.

degamma_lut_size_property

Optional CRTC property for the size of the degamma LUT as supported by the driver (read-only).

ctm_property

Optional CRTC property to set the matrix used to convert colors after the lookup in the degamma LUT.

gamma_lut_property

Optional CRTC property to set the LUT used to convert the colors, after the CTM matrix, to the gamma space of the connected screen.

gamma_lut_size_property

Optional CRTC property for the size of the gamma LUT as supported by the driver (read-only).

suggested_x_property

Optional connector property with a hint for the position of the output on the host's screen.

suggested_y_property

Optional connector property with a hint for the position of the output on the host's screen.

non_desktop_property

Optional connector property with a hint that device isn't a standard display, and the console/desktop, should not be displayed on it.

panel_orientation_property

Optional connector property indicating how the lcd-panel is mounted inside the casing (e.g. normal or upside-down).

writeback_fb_id_property

Property for writeback connectors, storing the ID of the output framebuffer. See also: [`drm_writeback_connector_init\(\)`](#)

writeback_pixel_formats_property

Property for writeback connectors, storing an array of the supported pixel formats for the writeback engine (read-only). See also: [`drm_writeback_connector_init\(\)`](#)

writeback_out_fence_ptr_property

Property for writeback connectors, fd pointer representing the outgoing fences for a writeback connector. Userspace should provide a pointer to a value of type s32, and then cast that pointer to u64. See also: [`drm_writeback_connector_init\(\)`](#)

hdr_output_metadata_property

Connector property containing hdr metadata. This will be provided by userspace compositors based on HDR content

content_protection_property

DRM ENUM property for content protection. See [*drm_connector_attach_content_protection*](#)

hdcp_content_type_property

DRM ENUM property for type of Protected Content.

preferred_depth

preferred RGB pixel depth, used by fb helpers

prefer_shadow

hint to userspace to prefer shadow-fb rendering

quirk_addfb_prefer_xbgr_30bpp

Special hack for legacy ADDFB to keep nouveau userspace happy. Should only ever be set by the nouveau kernel driver.

quirk_addfb_prefer_host_byte_order

When set to true `drm_mode_addfb()` will pick host byte order pixel format when calling `drm_mode_addfb2()`. This is how `drm_mode_addfb()` should have worked from day one. It didn't though, so we ended up with quirks in both kernel and userspace drivers to deal with the broken behavior. Simply fixing `drm_mode_addfb()` unconditionally would break these drivers, so add a quirk bit here to allow drivers opt-in.

async_page_flip

Does this device support async flips on the primary plane?

fb_modifiers_not_supported

When this flag is set, the DRM device will not expose modifier support to userspace. This is only used by legacy drivers that infer the buffer layout through heuristics without using modifiers. New drivers shall not set this flag.

normalize_zpos

If true the drm core will call [*drm_atomic_normalize_zpos\(\)*](#) as part of atomic mode checking from [*drm_atomic_helper_check\(\)*](#)

modifiers_property

Plane property to list support modifier/format combination.

cursor_width

hint to userspace for max cursor width

cursor_height

hint to userspace for max cursor height

suspend_state

Atomic state when suspended. Set by [*drm_mode_config_helper_suspend\(\)*](#) and cleared by [*drm_mode_config_helper_resume\(\)*](#).

helper_private

mid-layer private data

Description

Core mode resource tracking structure. All CRTC, encoders, and connectors enumerated by the driver are added here, as are global properties. Some global restrictions are also here, e.g. dimension restrictions.

Framebuffer sizes refer to the virtual screen that can be displayed by the CRTC. This can be different from the physical resolution programmed. The minimum width and height, stored in

min_width and **min_height**, describe the smallest size of the framebuffer. It correlates to the minimum programmable resolution. The maximum width, stored in **max_width**, is typically limited by the maximum pitch between two adjacent scanlines. The maximum height, stored in **max_height**, is usually only limited by the amount of addressable video memory. For hardware that has no real maximum, drivers should pick a reasonable default.

See also **DRM_SHADOW_PLANE_MAX_WIDTH** and **DRM_SHADOW_PLANE_MAX_HEIGHT**.

```
int drm_mode_config_init(struct drm_device *dev)
    DRM mode_configuration structure initialization
```

Parameters

```
struct drm_device *dev
    DRM device
```

Description

This is the unmanaged version of *drmm_mode_config_init()* for drivers which still explicitly call *drm_mode_config_cleanup()*.

FIXME: This function is deprecated and drivers should be converted over to *drmm_mode_config_init()*.

```
void drm_mode_config_reset(struct drm_device *dev)
    call ->reset callbacks
```

Parameters

```
struct drm_device *dev
    drm device
```

Description

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

```
int drmm_mode_config_init(struct drm_device *dev)
    managed DRM mode_configuration structure initialization
```

Parameters

```
struct drm_device *dev
    DRM device
```

Description

Initialize **dev**'s mode_config structure, used for tracking the graphics configuration of **dev**.

Since this initializes the modeset locks, no locking is possible. Which is no problem, since this should happen single threaded at init time. It is the driver's problem to ensure this guarantee.

Cleanup is automatically handled through registering *drm_mode_config_cleanup* with *drmm_add_action()*.

Return

0 on success, negative error value on failure.

void **drm_mode_config_cleanup**(struct *drm_device* *dev)
 free up DRM mode_config info

Parameters

struct drm_device *dev
 DRM device

Description

Free up all the connectors and CRTC's associated with this DRM device, then free up the frame-buffers and associated buffer objects.

Note that since this /should/ happen single-threaded at driver/device teardown time, no locking is required. It's the driver's job to ensure that this guarantee actually holds true.

FIXME: With the managed *drmm_mode_config_init()* it is no longer necessary for drivers to explicitly call this function.

4.3 Modeset Base Object Abstraction

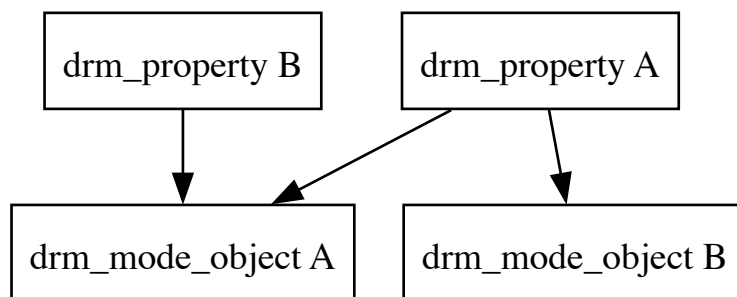


Fig. 3: Mode Objects and Properties

The base structure for all KMS objects is *struct drm_mode_object*. One of the base services it provides is tracking properties, which are especially important for the atomic IOCTL (see *Atomic Mode Setting*). The somewhat surprising part here is that properties are not directly instantiated on each object, but free-standing mode objects themselves, represented by *struct drm_property*, which only specify the type and value range of a property. Any given property can be attached multiple times to different objects using *drm_object_attach_property()*.

struct drm_mode_object
 base structure for modeset objects

Definition:

```
struct drm_mode_object {
    uint32_t id;
    uint32_t type;
    struct drm_object_properties *properties;
    struct kref refcount;
    void (*free_cb)(struct kref *kref);
};
```

Members

id
userspace visible identifier

type
type of the object, one of `DRM_MODE_OBJECT_*`

properties
properties attached to this object, including values

refcount
reference count for objects which with dynamic lifetime

free_cb
free function callback, only set for objects with dynamic lifetime

Description

Base structure for modeset objects visible to userspace. Objects can be looked up using `drm_mode_object_find()`. Besides basic uapi interface properties like **id** and **type** it provides two services:

- It tracks attached properties and their values. This is used by `drm_crtc`, `drm_plane` and `drm_connector`. Properties are attached by calling `drm_object_attach_property()` before the object is visible to userspace.
- For objects with dynamic lifetimes (as indicated by a non-NULL **free_cb**) it provides reference counting through `drm_mode_object_get()` and `drm_mode_object_put()`. This is used by `drm_framebuffer`, `drm_connector` and `drm_property_blob`. These objects provide specialized reference counting wrappers.

struct **drm_object_properties**
property tracking for `drm_mode_object`

Definition:

```
struct drm_object_properties {
    int count;
    struct drm_property *properties[DRM_OBJECT_MAX_PROPERTY];
    uint64_t values[DRM_OBJECT_MAX_PROPERTY];
};
```

Members

count
number of valid properties, must be less than or equal to `DRM_OBJECT_MAX_PROPERTY`.

properties
Array of pointers to `drm_property`.

NOTE: if we ever start dynamically destroying properties (ie. not at `drm_mode_config_cleanup()` time), then we'd have to do a better job of detaching property from mode objects to avoid dangling property pointers:

values
Array to store the property values, matching **properties**. Do not read/write values directly, but use `drm_object_property_get_value()` and `drm_object_property_set_value()`.

Note that atomic drivers do not store mutable properties in this array, but only the decoded values in the corresponding state structure. The decoding is done using the

`drm_crtc.atomic_get_property` and `drm_crtc.atomic_set_property` hooks for `struct drm_crtc`. For `struct drm_plane` the hooks are `drm_plane_funcs.atomic_get_property` and `drm_plane_funcs.atomic_set_property`. And for `struct drm_connector` the hooks are `drm_connector_funcs.atomic_get_property` and `drm_connector_funcs.atomic_set_property`.

Hence atomic drivers should not use `drm_object_property_set_value()` and `drm_object_property_get_value()` on mutable objects, i.e. those without the `DRM_MODE_PROP_IMMUTABLE` flag set.

For atomic drivers the default value of properties is stored in this array, so `drm_object_property_get_default_value` can be used to retrieve it.

struct `drm_mode_object` ***drm_mode_object_find**(struct `drm_device` *dev, struct `drm_file` *file_priv, uint32_t id, uint32_t type)

look up a drm object with static lifetime

Parameters

struct drm_device *dev
drm device

struct drm_file *file_priv
drm file

uint32_t id
id of the mode object

uint32_t type
type of the mode object

Description

This function is used to look up a modeset object. It will acquire a reference for reference counted objects. This reference must be dropped again by calling `drm_mode_object_put()`.

void **drm_mode_object_put**(struct `drm_mode_object` *obj)
release a mode object reference

Parameters

struct drm_mode_object *obj
DRM mode object

Description

This function decrements the object's refcount if it is a refcounted modeset object. It is a no-op on any other object. This is used to drop references acquired with `drm_mode_object_get()`.

void **drm_mode_object_get**(struct `drm_mode_object` *obj)
acquire a mode object reference

Parameters

struct drm_mode_object *obj
DRM mode object

Description

This function increments the object's refcount if it is a refcounted modeset object. It is a no-op on any other object. References should be dropped again by calling `drm_mode_object_put()`.

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

attach a property to a modeset object

Parameters

struct *drm_mode_object* *obj
drm modeset object

struct *drm_property* *property
property to attach

uint64_t init_val
initial value of the property

Description

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

Note that all properties must be attached before the object itself is registered and accessible from userspace.

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

set the value of a property

Parameters

struct *drm_mode_object* *obj
drm mode object to set property value for

struct *drm_property* *property
property to set

uint64_t val
value the property should be set to

Description

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

Note that atomic drivers should not have any need to call this, the core will ensure consistency of values reported back to userspace through the appropriate ->atomic_get_property callback. Only legacy drivers should call this function to update the tracked value (after clamping and other restrictions have been applied).

Return

Zero on success, error code on failure.

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

retrieve the value of a property

Parameters

struct *drm_mode_object* *obj
drm mode object to get property value from

struct drm_property *property

property to retrieve

uint64_t *val

storage for the property value

Description

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

Atomic drivers should never call this function directly, the core will read out property values through the various `->atomic_get_property` callbacks.

Return

Zero on success, error code on failure.

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

retrieve the default value of a property when in atomic mode.

Parameters

struct drm_mode_object *obj

drm mode object to get property value from

struct drm_property *property

property to retrieve

uint64_t *val

storage for the property value

Description

This function retrieves the default state of the given property as passed in to `drm_object_attach_property`

Only atomic drivers should call this function directly, as for non-atomic drivers it will return the current value.

Return

Zero on success, error code on failure.

4.4 Atomic Mode Setting

Atomic provides transactional modeset (including planes) updates, but a bit differently from the usual transactional approach of try-commit and rollback:

- Firstly, no hardware changes are allowed when the commit would fail. This allows us to implement the `DRM_MODE_ATOMIC_TEST_ONLY` mode, which allows userspace to explore whether certain configurations would work or not.
- This would still allow setting and rollback of just the software state, simplifying conversion of existing drivers. But auditing drivers for correctness of the `atomic_check` code becomes really hard with that: Rolling back changes in data structures all over the place is hard to get right.

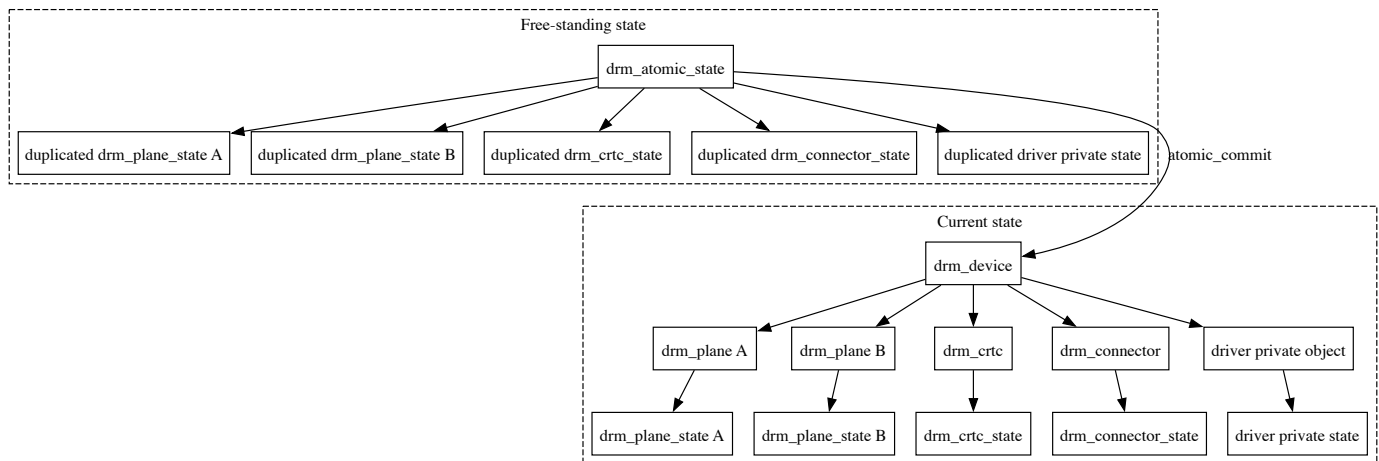


Fig. 4: Mode Objects and Properties

- Lastly, for backwards compatibility and to support all use-cases, atomic updates need to be incremental and be able to execute in parallel. Hardware doesn't always allow it, but where possible plane updates on different CRTC's should not interfere, and not get stalled due to output routing changing on different CRTC's.

Taken all together there's two consequences for the atomic design:

- The overall state is split up into per-object state structures: *struct drm_plane_state* for planes, *struct drm_crtc_state* for CRTC's and *struct drm_connector_state* for connectors. These are the only objects with userspace-visible and settable state. For internal state drivers can subclass these structures through embedding, or add entirely new state structures for their globally shared hardware functions, see *struct drm_private_state*.
- An atomic update is assembled and validated as an entirely free-standing pile of structures within the *drm_atomic_state* container. Driver private state structures are also tracked in the same structure; see the next chapter. Only when a state is committed is it applied to the driver and modeset objects. This way rolling back an update boils down to releasing memory and unreferencing objects like framebuffers.

Locking of atomic state structures is internally using *struct drm_modeset_lock*. As a general rule the locking shouldn't be exposed to drivers, instead the right locks should be automatically acquired by any function that duplicates or peeks into a state, like e.g. *drm_atomic_get_crtc_state()*. Locking only protects the software data structure, ordering of committing state changes to hardware is sequenced using *struct drm_crtc_commit*.

Read on in this chapter, and also in *Atomic Modeset Helper Functions Reference* for more detailed coverage of specific topics.

4.4.1 Handling Driver Private State

Very often the DRM objects exposed to userspace in the atomic modeset api (*drm_connector*, *drm_crtc* and *drm_plane*) do not map neatly to the underlying hardware. Especially for any kind of shared resources (e.g. shared clocks, scaler units, bandwidth and fifo limits shared among a group of planes or CRTC's, and so on) it makes sense to model these as independent objects. Drivers then need to do similar state tracking and commit ordering for such private (since not exposed to userspace) objects as the atomic core and helpers already provide for connectors, planes and CRTC's.

To make this easier on drivers the atomic core provides some support to track driver private state objects using struct *drm_private_obj*, with the associated state struct *drm_private_state*.

Similar to userspace-exposed objects, private state structures can be acquired by calling *drm_atomic_get_private_obj_state()*. This also takes care of locking, hence drivers should not have a need to call *drm_modeset_lock()* directly. Sequence of the actual hardware state commit is not handled, drivers might need to keep track of *struct drm_crtc_commit* within subclassed structure of *drm_private_state* as necessary, e.g. similar to *drm_plane_state.commit*. See also *drm_atomic_state.fake_commit*.

All private state structures contained in a *drm_atomic_state* update can be iterated using *for_each_oldnew_private_obj_in_state()*, *for_each_new_private_obj_in_state()* and *for_each_old_private_obj_in_state()*. Drivers are recommended to wrap these for each type of driver private state object they have, filtering on *drm_private_obj.funcs* using *for_each_if()*, at least if they want to iterate over all objects of a given type.

An earlier way to handle driver private state was by subclassing struct *drm_atomic_state*. But since that encourages non-standard ways to implement the check/commit split atomic requires (by using e.g. “check and rollback or commit instead” of “duplicate state, check, then either commit or release duplicated state”) it is deprecated in favour of using *drm_private_state*.

4.4.2 Atomic Mode Setting Function Reference

struct **drm_crtc_commit**

track modeset commits on a CRTC

Definition:

```
struct drm_crtc_commit {
    struct drm_crtc *crtc;
    struct kref ref;
    struct completion flip_done;
    struct completion hw_done;
    struct completion cleanup_done;
    struct list_head commit_entry;
    struct drm_pending_vblank_event *event;
    bool abort_completion;
};
```

Members

crtc

DRM CRTC for this commit.

ref

Reference count for this structure. Needed to allow blocking on completions without the risk of the completion disappearing meanwhile.

flip_done

Will be signalled when the hardware has flipped to the new set of buffers. Signals at the same time as when the drm event for this commit is sent to userspace, or when an out-fence is signalled. Note that for most hardware, in most cases this happens after **hw_done** is signalled.

Completion of this stage is signalled implicitly by calling `drm_crtc_send_vblank_event()` on `drm_crtc_state.event`.

hw_done

Will be signalled when all hw register changes for this commit have been written out. Especially when disabling a pipe this can be much later than **flip_done**, since that can signal already when the screen goes black, whereas to fully shut down a pipe more register I/O is required.

Note that this does not need to include separately reference-counted resources like backing storage buffer pinning, or runtime pm management.

Drivers should call `drm_atomic_helper_commit_hw_done()` to signal completion of this stage.

cleanup_done

Will be signalled after old buffers have been cleaned up by calling `drm_atomic_helper_cleanup_planes()`. Since this can only happen after a vblank wait completed it might be a bit later. This completion is useful to throttle updates and avoid hardware updates getting ahead of the buffer cleanup too much.

Drivers should call `drm_atomic_helper_commit_cleanup_done()` to signal completion of this stage.

commit_entry

Entry on the per-CRTC `drm_crtc.commit_list`. Protected by `$drm_crtc.commit_lock`.

event

`drm_pending_vblank_event` pointer to clean up private events.

abort_completion

A flag that's set after `drm_atomic_helper_setup_commit()` takes a second reference for the completion of `$drm_crtc_state.event`. It's used by the free code to remove the second reference if commit fails.

Description

This structure is used to track pending modeset changes and atomic commit on a per-CRTC basis. Since updating the list should never block, this structure is reference counted to allow waiters to safely wait on an event to complete, without holding any locks.

It has 3 different events in total to allow a fine-grained synchronization between outstanding updates:

atomic commit thread		hardware
write new state into hardware	---->	...
signal hw_done		

```
...                               switch to new state on next
                                v/hblank

wait for buffers to show up      ...

...                               send completion irq
                                irq handler signals flip_done

cleanup old buffers

signal cleanup_done

wait for flip_done               <----
clean up atomic state
```

The important bit to know is that `cleanup_done` is the terminal event, but the ordering between `flip_done` and `hw_done` is entirely up to the specific driver and modeset state change.

For an implementation of how to use this look at [`drm_atomic_helper_setup_commit\(\)`](#) from the atomic helper library.

See also [`drm_crtc_commit_wait\(\)`](#).

struct **drm_private_state_funcs**

atomic state functions for private objects

Definition:

```
struct drm_private_state_funcs {
    struct drm_private_state *(*atomic_duplicate_state)(struct drm_private_obj
→*obj);
    void (*atomic_destroy_state)(struct drm_private_obj *obj, struct drm_
→private_state *state);
    void (*atomic_print_state)(struct drm_printer *p, const struct drm_private_
→state *state);
};
```

Members

atomic_duplicate_state

Duplicate the current state of the private object and return it. It is an error to call this before `obj->state` has been initialized.

RETURNS:

Duplicated atomic state or NULL when `obj->state` is not initialized or allocation failed.

atomic_destroy_state

Frees the private object state created with **atomic_duplicate_state**.

atomic_print_state

If driver subclasses [struct `drm_private_state`](#), it should implement this optional hook for printing additional driver specific state.

Do not call this directly, use `drm_atomic_private_obj_print_state()` instead.

Description

These hooks are used by atomic helpers to create, swap and destroy states of private objects. The structure itself is used as a vtable to identify the associated private object type. Each private object type that needs to be added to the atomic states is expected to have an implementation of these hooks and pass a pointer to its `drm_private_state_funcs` struct to `drm_atomic_get_private_obj_state()`.

struct **drm_private_obj**

base struct for driver private atomic object

Definition:

```
struct drm_private_obj {
    struct list_head head;
    struct drm_modeset_lock lock;
    struct drm_private_state *state;
    const struct drm_private_state_funcs *funcs;
};
```

Members

head

List entry used to attach a private object to a *drm_device* (queued to *drm_mode_config.privobj_list*).

lock

Modeset lock to protect the state object.

state

Current atomic state for this driver private object.

funcs

Functions to manipulate the state of this driver private object, see *drm_private_state_funcs*.

Description

A driver private object is initialized by calling *drm_atomic_private_obj_init()* and cleaned up by calling *drm_atomic_private_obj_fini()*.

Currently only tracks the state update functions and the opaque driver private state itself, but in the future might also track which *drm_modeset_lock* is required to duplicate and update this object's state.

All private objects must be initialized before the DRM device they are attached to is registered to the DRM subsystem (call to *drm_dev_register()*) and should stay around until this DRM device is unregistered (call to *drm_dev_unregister()*). In other words, private objects lifetime is tied to the DRM device lifetime. This implies that:

1/ all calls to *drm_atomic_private_obj_init()* must be done before calling *drm_dev_register()*

2/ all calls to *drm_atomic_private_obj_fini()* must be done after calling *drm_dev_unregister()*

If that private object is used to store a state shared by multiple CRTC's, proper care must be taken to ensure that non-blocking commits are properly ordered to avoid a use-after-free issue.

Indeed, assuming a sequence of two non-blocking `drm_atomic_commit` on two different `drm_crtc` using different `drm_plane` and `drm_connector`, so with no resources shared, there's no guarantee on which commit is going to happen first. However, the second `drm_atomic_commit` will consider the first `drm_private_obj` its old state, and will be in charge of freeing it whenever the second `drm_atomic_commit` is done.

If the first `drm_atomic_commit` happens after it, it will consider its `drm_private_obj` the new state and will be likely to access it, resulting in an access to a freed memory region. Drivers should store (and get a reference to) the `drm_crtc_commit` structure in our private state in `drm_mode_config_helper_funcs.atomic_commit_setup`, and then wait for that commit to complete as the first step of `drm_mode_config_helper_funcs.atomic_commit_tail`, similar to `drm_atomic_helper_wait_for_dependencies()`.

drm_for_each_privobj

`drm_for_each_privobj (privobj, dev)`

private object iterator

Parameters

privobj

pointer to the current private object. Updated after each iteration

dev

the DRM device we want get private objects from

Description

Allows one to iterate over all private objects attached to **dev**

struct **drm_private_state**

base struct for driver private object state

Definition:

```
struct drm_private_state {
    struct drm_atomic_state *state;
    struct drm_private_obj *obj;
};
```

Members

state

backpointer to global `drm_atomic_state`

obj

backpointer to the private object

Description

Currently only contains a backpointer to the overall atomic update, and the relevant private object but in the future also might hold synchronization information similar to e.g. `drm_crtc.commit`.

struct **drm_atomic_state**

the global state object for atomic updates

Definition:

```

struct drm_atomic_state {
    struct kref ref;
    struct drm_device *dev;
    bool allow_modeset : 1;
    bool legacy_cursor_update : 1;
    bool async_update : 1;
    bool duplicated : 1;
    struct __drm_planes_state *planes;
    struct __drm_crtcs_state *crtcs;
    int num_connector;
    struct __drm_connectors_state *connectors;
    int num_private_objs;
    struct __drm_private_objs_state *private_objs;
    struct drm_modeset_acquire_ctx *acquire_ctx;
    struct drm_crtc_commit *fake_commit;
    struct work_struct commit_work;
};

```

Members

ref

count of all references to this state (will not be freed until zero)

dev

parent DRM device

allow_modeset

Allow full modeset. This is used by the ATOMIC IOCTL handler to implement the `DRM_MODE_ATOMIC_ALLOW_MODESET` flag. Drivers should never consult this flag, instead looking at the output of [drm_atomic_crtc_needs_modeset\(\)](#).

legacy_cursor_update

Hint to enforce legacy cursor IOCTL semantics.

WARNING: This is thoroughly broken and pretty much impossible to implement correctly. Drivers must ignore this and should instead implement [drm_plane_helper_funcs.atomic_async_check](#) and [drm_plane_helper_funcs.atomic_async_commit](#) hooks. New users of this flag are not allowed.

async_update

hint for asynchronous plane update

duplicated

Indicates whether or not this atomic state was duplicated using [drm_atomic_helper_duplicate_state\(\)](#). Drivers and atomic helpers should use this to fixup normal inconsistencies in duplicated states.

planes

pointer to array of structures with per-plane data

crtcs

pointer to array of CRTC pointers

num_connector

size of the **connectors** and **connector_states** arrays

connectors

pointer to array of structures with per-connector data

num_private_objs

size of the **private_objs** array

private_objs

pointer to array of private object pointers

acquire_ctx

acquire context for this atomic modeset state update

fake_commit

Used for signaling unbound planes/connectors. When a connector or plane is not bound to any CRTC, it's still important to preserve linearity to prevent the atomic states from being freed to early.

This commit (if set) is not bound to any CRTC, but will be completed when [`drm_atomic_helper_commit_hw_done\(\)`](#) is called.

commit_work

Work item which can be used by the driver or helpers to execute the commit without blocking.

Description

States are added to an atomic update by calling [`drm_atomic_get_crtc_state\(\)`](#), [`drm_atomic_get_plane_state\(\)`](#), [`drm_atomic_get_connector_state\(\)`](#), or for private state structures, [`drm_atomic_get_private_obj_state\(\)`](#).

struct [`drm_crtc_commit`](#) ***drm_crtc_commit_get**(struct [`drm_crtc_commit`](#) *commit)
acquire a reference to the CRTC commit

Parameters

struct [`drm_crtc_commit`](#) *commit
CRTC commit

Description

Increases the reference of **commit**.

Return

The pointer to **commit**, with reference increased.

void **drm_crtc_commit_put**(struct [`drm_crtc_commit`](#) *commit)
release a reference to the CRTC commit

Parameters

struct [`drm_crtc_commit`](#) *commit
CRTC commit

Description

This releases a reference to **commit** which is freed after removing the final reference. No locking required and callable from any context.

struct [`drm_atomic_state`](#) ***drm_atomic_state_get**(struct [`drm_atomic_state`](#) *state)
acquire a reference to the atomic state

Parameters

struct drm_atomic_state *state
The atomic state

Description

Returns a new reference to the **state**

void **drm_atomic_state_put**(struct *drm_atomic_state* *state)
release a reference to the atomic state

Parameters

struct drm_atomic_state *state
The atomic state

Description

This releases a reference to **state** which is freed after removing the final reference. No locking required and callable from any context.

struct *drm_crtc_state* ***drm_atomic_get_existing_crtc_state**(const struct *drm_atomic_state* *state,
struct *drm_crtc* *crtc)

get CRTC state, if it exists

Parameters

const struct drm_atomic_state *state
global atomic state object

struct drm_crtc *crtc
CRTC to grab

Description

This function returns the CRTC state for the given CRTC, or NULL if the CRTC is not part of the global atomic state.

This function is deprecated, **drm_atomic_get_old_crtc_state** or **drm_atomic_get_new_crtc_state** should be used instead.

struct *drm_crtc_state* ***drm_atomic_get_old_crtc_state**(const struct *drm_atomic_state* *state, struct *drm_crtc* *crtc)

get old CRTC state, if it exists

Parameters

const struct drm_atomic_state *state
global atomic state object

struct drm_crtc *crtc
CRTC to grab

Description

This function returns the old CRTC state for the given CRTC, or NULL if the CRTC is not part of the global atomic state.

struct *drm_crtc_state* ***drm_atomic_get_new_crtc_state**(const struct *drm_atomic_state* *state, struct *drm_crtc* *crtc)

get new CRTC state, if it exists

Parameters

const struct drm_atomic_state *state
global atomic state object

struct drm_crtc *crtc
CRTC to grab

Description

This function returns the new CRTC state for the given CRTC, or NULL if the CRTC is not part of the global atomic state.

struct *drm_plane_state* ***drm_atomic_get_existing_plane_state**(const struct *drm_atomic_state* *state, struct *drm_plane* *plane)

get plane state, if it exists

Parameters

const struct drm_atomic_state *state
global atomic state object

struct drm_plane *plane
plane to grab

Description

This function returns the plane state for the given plane, or NULL if the plane is not part of the global atomic state.

This function is deprecated, **drm_atomic_get_old_plane_state** or **drm_atomic_get_new_plane_state** should be used instead.

struct *drm_plane_state* ***drm_atomic_get_old_plane_state**(const struct *drm_atomic_state* *state, struct *drm_plane* *plane)

get plane state, if it exists

Parameters

const struct drm_atomic_state *state
global atomic state object

struct drm_plane *plane
plane to grab

Description

This function returns the old plane state for the given plane, or NULL if the plane is not part of the global atomic state.

struct *drm_plane_state* ***drm_atomic_get_new_plane_state**(const struct *drm_atomic_state* *state, struct *drm_plane* *plane)

get plane state, if it exists

Parameters

const struct drm_atomic_state *state

global atomic state object

struct drm_plane *plane

plane to grab

Description

This function returns the new plane state for the given plane, or NULL if the plane is not part of the global atomic state.

struct *drm_connector_state* ***drm_atomic_get_existing_connector_state**(const struct *drm_atomic_state* *state, struct *drm_connector* *connector)

get connector state, if it exists

Parameters

const struct drm_atomic_state *state

global atomic state object

struct drm_connector *connector

connector to grab

Description

This function returns the connector state for the given connector, or NULL if the connector is not part of the global atomic state.

This function is deprecated, **drm_atomic_get_old_connector_state** or **drm_atomic_get_new_connector_state** should be used instead.

struct *drm_connector_state* ***drm_atomic_get_old_connector_state**(const struct *drm_atomic_state* *state, struct *drm_connector* *connector)

get connector state, if it exists

Parameters

const struct drm_atomic_state *state

global atomic state object

struct drm_connector *connector

connector to grab

Description

This function returns the old connector state for the given connector, or NULL if the connector is not part of the global atomic state.

struct *drm_connector_state* ***drm_atomic_get_new_connector_state**(const struct *drm_atomic_state* *state, struct *drm_connector* *connector)

get connector state, if it exists

Parameters

const struct drm_atomic_state *state

global atomic state object

struct drm_connector *connector

connector to grab

Description

This function returns the new connector state for the given connector, or NULL if the connector is not part of the global atomic state.

```
const struct drm_plane_state *__drm_atomic_get_current_plane_state(const struct
                                                                    drm_atomic_state
                                                                    *state, struct
                                                                    drm_plane *plane)
```

get current plane state

Parameters

const struct drm_atomic_state *state

global atomic state object

struct drm_plane *plane

plane to grab

Description

This function returns the plane state for the given plane, either from **state**, or if the plane isn't part of the atomic state update, from **plane**. This is useful in atomic check callbacks, when drivers need to peek at, but not change, state of other planes, since it avoids threading an error code back up the call chain.

WARNING:

Note that this function is in general unsafe since it doesn't check for the required locking for access state structures. Drivers must ensure that it is safe to access the returned state structure through other means. One common example is when planes are fixed to a single CRTC, and the driver knows that the CRTC lock is held already. In that case holding the CRTC lock gives a read-lock on all planes connected to that CRTC. But if planes can be reassigned things get more tricky. In that case it's better to use `drm_atomic_get_plane_state` and wire up full error handling.

Read-only pointer to the current plane state.

Return

for_each_oldnew_connector_in_state

```
for_each_oldnew_connector_in_state (__state, connector, old_connector_state,
new_connector_state, __i)
```

iterate over all connectors in an atomic update

Parameters

__state*struct drm_atomic_state* pointer**connector***struct drm_connector* iteration cursor**old_connector_state***struct drm_connector_state* iteration cursor for the old state**new_connector_state***struct drm_connector_state* iteration cursor for the new state**__i**

int iteration cursor, for macro-internal use

Description

This iterates over all connectors in an atomic update, tracking both old and new state. This is useful in places where the state delta needs to be considered, for example in atomic check functions.

for_each_old_connector_in_state

`for_each_old_connector_in_state (__state, connector, old_connector_state, __i)`
iterate over all connectors in an atomic update

Parameters**__state***struct drm_atomic_state* pointer**connector***struct drm_connector* iteration cursor**old_connector_state***struct drm_connector_state* iteration cursor for the old state**__i**

int iteration cursor, for macro-internal use

Description

This iterates over all connectors in an atomic update, tracking only the old state. This is useful in disable functions, where we need the old state the hardware is still in.

for_each_new_connector_in_state

`for_each_new_connector_in_state (__state, connector, new_connector_state, __i)`
iterate over all connectors in an atomic update

Parameters**__state***struct drm_atomic_state* pointer**connector***struct drm_connector* iteration cursor**new_connector_state***struct drm_connector_state* iteration cursor for the new state

__i
int iteration cursor, for macro-internal use

Description

This iterates over all connectors in an atomic update, tracking only the new state. This is useful in enable functions, where we need the new state the hardware should be in when the atomic commit operation has completed.

for_each_oldnew_crtc_in_state

for_each_oldnew_crtc_in_state (__state, crtc, old_crtc_state, new_crtc_state, __i)

iterate over all CRTC's in an atomic update

Parameters

__state
struct drm_atomic_state pointer

crtc
struct drm_crtc iteration cursor

old_crtc_state
struct drm_crtc_state iteration cursor for the old state

new_crtc_state
struct drm_crtc_state iteration cursor for the new state

__i
int iteration cursor, for macro-internal use

Description

This iterates over all CRTC's in an atomic update, tracking both old and new state. This is useful in places where the state delta needs to be considered, for example in atomic check functions.

for_each_old_crtc_in_state

for_each_old_crtc_in_state (__state, crtc, old_crtc_state, __i)

iterate over all CRTC's in an atomic update

Parameters

__state
struct drm_atomic_state pointer

crtc
struct drm_crtc iteration cursor

old_crtc_state
struct drm_crtc_state iteration cursor for the old state

__i
int iteration cursor, for macro-internal use

Description

This iterates over all CRTC's in an atomic update, tracking only the old state. This is useful in disable functions, where we need the old state the hardware is still in.

for_each_new_crtc_in_state

`for_each_new_crtc_in_state (__state, crtc, new_crtc_state, __i)`
iterate over all CRTC's in an atomic update

Parameters**__state***struct drm_atomic_state* pointer**crtc***struct drm_crtc* iteration cursor**new_crtc_state***struct drm_crtc_state* iteration cursor for the new state**__i**

int iteration cursor, for macro-internal use

Description

This iterates over all CRTC's in an atomic update, tracking only the new state. This is useful in enable functions, where we need the new state the hardware should be in when the atomic commit operation has completed.

for_each_oldnew_plane_in_state

`for_each_oldnew_plane_in_state (__state, plane, old_plane_state, new_plane_state, __i)`
iterate over all planes in an atomic update

Parameters**__state***struct drm_atomic_state* pointer**plane***struct drm_plane* iteration cursor**old_plane_state***struct drm_plane_state* iteration cursor for the old state**new_plane_state***struct drm_plane_state* iteration cursor for the new state**__i**

int iteration cursor, for macro-internal use

Description

This iterates over all planes in an atomic update, tracking both old and new state. This is useful in places where the state delta needs to be considered, for example in atomic check functions.

for_each_oldnew_plane_in_state_reverse

`for_each_oldnew_plane_in_state_reverse (__state, plane, old_plane_state, new_plane_state, __i)`
iterate over all planes in an atomic update in reverse order

Parameters

__state

struct drm_atomic_state pointer

plane

struct drm_plane iteration cursor

old_plane_state

struct drm_plane_state iteration cursor for the old state

new_plane_state

struct drm_plane_state iteration cursor for the new state

__i

int iteration cursor, for macro-internal use

Description

This iterates over all planes in an atomic update in reverse order, tracking both old and new state. This is useful in places where the state delta needs to be considered, for example in atomic check functions.

for_each_new_plane_in_state_reverse

for_each_new_plane_in_state_reverse (__state, plane, new_plane_state, __i)

other than only tracking new state, it's the same as
for_each_oldnew_plane_in_state_reverse

Parameters

__state

struct drm_atomic_state pointer

plane

struct drm_plane iteration cursor

new_plane_state

struct drm_plane_state iteration cursor for the new state

__i

int iteration cursor, for macro-internal use

for_each_old_plane_in_state

for_each_old_plane_in_state (__state, plane, old_plane_state, __i)

iterate over all planes in an atomic update

Parameters

__state

struct drm_atomic_state pointer

plane

struct drm_plane iteration cursor

old_plane_state

struct drm_plane_state iteration cursor for the old state

__i

int iteration cursor, for macro-internal use

Description

This iterates over all planes in an atomic update, tracking only the old state. This is useful in disable functions, where we need the old state the hardware is still in.

for_each_new_plane_in_state

`for_each_new_plane_in_state (__state, plane, new_plane_state, __i)`

iterate over all planes in an atomic update

Parameters

__state

struct drm_atomic_state pointer

plane

struct drm_plane iteration cursor

new_plane_state

struct drm_plane_state iteration cursor for the new state

__i

int iteration cursor, for macro-internal use

Description

This iterates over all planes in an atomic update, tracking only the new state. This is useful in enable functions, where we need the new state the hardware should be in when the atomic commit operation has completed.

for_each_oldnew_private_obj_in_state

`for_each_oldnew_private_obj_in_state (__state, obj, old_obj_state, new_obj_state, __i)`

iterate over all private objects in an atomic update

Parameters

__state

struct drm_atomic_state pointer

obj

struct drm_private_obj iteration cursor

old_obj_state

struct drm_private_state iteration cursor for the old state

new_obj_state

struct drm_private_state iteration cursor for the new state

__i

int iteration cursor, for macro-internal use

Description

This iterates over all private objects in an atomic update, tracking both old and new state. This is useful in places where the state delta needs to be considered, for example in atomic check functions.

for_each_old_private_obj_in_state

`for_each_old_private_obj_in_state (__state, obj, old_obj_state, __i)`
iterate over all private objects in an atomic update

Parameters

__state

struct drm_atomic_state pointer

obj

struct drm_private_obj iteration cursor

old_obj_state

struct drm_private_state iteration cursor for the old state

__i

int iteration cursor, for macro-internal use

Description

This iterates over all private objects in an atomic update, tracking only the old state. This is useful in disable functions, where we need the old state the hardware is still in.

for_each_new_private_obj_in_state

`for_each_new_private_obj_in_state (__state, obj, new_obj_state, __i)`
iterate over all private objects in an atomic update

Parameters

__state

struct drm_atomic_state pointer

obj

struct drm_private_obj iteration cursor

new_obj_state

struct drm_private_state iteration cursor for the new state

__i

int iteration cursor, for macro-internal use

Description

This iterates over all private objects in an atomic update, tracking only the new state. This is useful in enable functions, where we need the new state the hardware should be in when the atomic commit operation has completed.

bool **drm_atomic_crtc_needs_modeset**(const struct *drm_crtc_state* *state)
compute combined modeset need

Parameters

const struct *drm_crtc_state* *state

drm_crtc_state for the CRTC

Description

To give drivers flexibility `struct drm_crtc_state` has 3 booleans to track whether the state CRTC changed enough to need a full modeset cycle: `mode_changed`, `active_changed` and `connectors_changed`. This helper simply combines these three to compute the overall need for a modeset for **state**.

The atomic helper code sets these booleans, but drivers can and should change them appropriately to accurately represent whether a modeset is really needed. In general, drivers should avoid full modesets whenever possible.

For example if the CRTC mode has changed, and the hardware is able to enact the requested mode change without going through a full modeset, the driver should clear `mode_changed` in its `drm_mode_config_funcs.atomic_check` implementation.

```
bool drm_atomic_crtc_effectively_active(const struct drm_crtc_state *state)
    compute whether CRTC is actually active
```

Parameters

```
const struct drm_crtc_state *state
    drm_crtc_state for the CRTC
```

Description

When in self refresh mode, the `crtc_state->active` value will be false, since the CRTC is off. However in some cases we're interested in whether the CRTC is active, or effectively active (ie: it's connected to an active display). In these cases, use this function instead of just checking active.

```
struct drm_bus_cfg
    bus configuration
```

Definition:

```
struct drm_bus_cfg {
    u32 format;
    u32 flags;
};
```

Members

format

format used on this bus (one of the `MEDIA_BUS_FMT_*` format)

This field should not be directly modified by drivers (`drm_atomic_bridge_chain_select_bus_fmts()` takes care of the bus format negotiation).

flags

`DRM_BUS_*` flags used on this bus

Description

This structure stores the configuration of a physical bus between two components in an output pipeline, usually between two bridges, an encoder and a bridge, or a bridge and a connector.

The bus configuration is stored in `drm_bridge_state` separately for the input and output buses, as seen from the point of view of each bridge. The bus configuration of a bridge output is usually identical to the configuration of the next bridge's input, but may differ if the signals are modified between the two bridges, for instance by an inverter on the board. The input and

output configurations of a bridge may differ if the bridge modifies the signals internally, for instance by performing format conversion, or modifying signals polarities.

struct **drm_bridge_state**

Atomic bridge state object

Definition:

```
struct drm_bridge_state {
    struct drm_private_state base;
    struct drm_bridge *bridge;
    struct drm_bus_cfg input_bus_cfg;
    struct drm_bus_cfg output_bus_cfg;
};
```

Members

base

inherit from *drm_private_state*

bridge

the bridge this state refers to

input_bus_cfg

input bus configuration

output_bus_cfg

input bus configuration

int **drm_crtc_commit_wait**(struct *drm_crtc_commit* *commit)

Waits for a commit to complete

Parameters

struct *drm_crtc_commit* *commit

drm_crtc_commit to wait for

Description

Waits for a given *drm_crtc_commit* to be programmed into the hardware and flipped to.

0 on success, a negative error code otherwise.

Return

void **drm_atomic_state_default_release**(struct *drm_atomic_state* *state)

release memory initialized by *drm_atomic_state_init*

Parameters

struct *drm_atomic_state* *state

atomic state

Description

Free all the memory allocated by *drm_atomic_state_init*. This should only be used by drivers which are still subclassing *drm_atomic_state* and haven't switched to *drm_private_state* yet.

int **drm_atomic_state_init**(struct *drm_device* *dev, struct *drm_atomic_state* *state)

init new atomic state

Parameters

struct drm_device *dev
DRM device

struct drm_atomic_state *state
atomic state

Description

Default implementation for filling in a new atomic state. This should only be used by drivers which are still subclassing *drm_atomic_state* and haven't switched to *drm_private_state* yet.

struct *drm_atomic_state* ***drm_atomic_state_alloc**(struct *drm_device* *dev)
allocate atomic state

Parameters

struct drm_device *dev
DRM device

Description

This allocates an empty atomic state to track updates.

void **drm_atomic_state_default_clear**(struct *drm_atomic_state* *state)
clear base atomic state

Parameters

struct drm_atomic_state *state
atomic state

Description

Default implementation for clearing atomic state. This should only be used by drivers which are still subclassing *drm_atomic_state* and haven't switched to *drm_private_state* yet.

void **drm_atomic_state_clear**(struct *drm_atomic_state* *state)
clear state object

Parameters

struct drm_atomic_state *state
atomic state

Description

When the w/w mutex algorithm detects a deadlock we need to back off and drop all locks. So someone else could sneak in and change the current modeset configuration. Which means that all the state assembled in **state** is no longer an atomic update to the current state, but to some arbitrary earlier state. Which could break assumptions the driver's *drm_mode_config_funcs.atomic_check* likely relies on.

Hence we must clear all cached state and completely start over, using this function.

void **__drm_atomic_state_free**(struct kref *ref)
free all memory for an atomic state

Parameters

struct kref *ref

This atomic state to deallocate

Description

This frees all memory associated with an atomic state, including all the per-object state for planes, CRTC's and connectors.

struct *drm_crtc_state* ***drm_atomic_get_crtc_state**(struct *drm_atomic_state* *state, struct *drm_crtc* *crtc)

get CRTC state

Parameters

struct drm_atomic_state *state

global atomic state object

struct drm_crtc *crtc

CRTC to get state object for

Description

This function returns the CRTC state for the given CRTC, allocating it if needed. It will also grab the relevant CRTC lock to make sure that the state is consistent.

WARNING: Drivers may only add new CRTC states to a **state** if `drm_atomic_state.allow_modeset` is set, or if it's a driver-internal commit not created by userspace through an IOCTL call.

Either the allocated state or the error code encoded into the pointer. When the error is `EDEADLK` then the w/w mutex code has detected a deadlock and the entire atomic sequence must be restarted. All other errors are fatal.

Return

struct *drm_plane_state* ***drm_atomic_get_plane_state**(struct *drm_atomic_state* *state, struct *drm_plane* *plane)

get plane state

Parameters

struct drm_atomic_state *state

global atomic state object

struct drm_plane *plane

plane to get state object for

Description

This function returns the plane state for the given plane, allocating it if needed. It will also grab the relevant plane lock to make sure that the state is consistent.

Either the allocated state or the error code encoded into the pointer. When the error is `EDEADLK` then the w/w mutex code has detected a deadlock and the entire atomic sequence must be restarted. All other errors are fatal.

Return

void **drm_atomic_private_obj_init**(struct *drm_device* *dev, struct *drm_private_obj* *obj, struct *drm_private_state* *state, const struct *drm_private_state_funcs* *funcs)

initialize private object

Parameters

struct *drm_device* *dev

DRM device this object will be attached to

struct *drm_private_obj* *obj

private object

struct *drm_private_state* *state

initial private object state

const struct *drm_private_state_funcs* *funcs

pointer to the struct of function pointers that identify the object type

Description

Initialize the private object, which can be embedded into any driver private object that needs its own atomic state.

void **drm_atomic_private_obj_fini**(struct *drm_private_obj* *obj)

finalize private object

Parameters

struct *drm_private_obj* *obj

private object

Description

Finalize the private object.

struct *drm_private_state* ***drm_atomic_get_private_obj_state**(struct *drm_atomic_state* *state, struct *drm_private_obj* *obj)

get private object state

Parameters

struct *drm_atomic_state* *state

global atomic state

struct *drm_private_obj* *obj

private object to get the state for

Description

This function returns the private object state for the given private object, allocating the state if needed. It will also grab the relevant private object lock to make sure that the state is consistent.

Either the allocated state or the error code encoded into a pointer.

Return

struct *drm_private_state* ***drm_atomic_get_old_private_obj_state**(const struct *drm_atomic_state* *state, struct *drm_private_obj* *obj)

Parameters

const struct drm_atomic_state *state
global atomic state object

struct drm_private_obj *obj
private_obj to grab

Description

This function returns the old private object state for the given private_obj, or NULL if the private_obj is not part of the global atomic state.

struct *drm_private_state* ***drm_atomic_get_new_private_obj_state**(const struct *drm_atomic_state* *state, struct *drm_private_obj* *obj)

Parameters

const struct drm_atomic_state *state
global atomic state object

struct drm_private_obj *obj
private_obj to grab

Description

This function returns the new private object state for the given private_obj, or NULL if the private_obj is not part of the global atomic state.

struct *drm_connector* ***drm_atomic_get_old_connector_for_encoder**(const struct *drm_atomic_state* *state, struct *drm_encoder* *encoder)

Get old connector for an encoder

Parameters

const struct drm_atomic_state *state
Atomic state

struct drm_encoder *encoder
The encoder to fetch the connector state for

Description

This function finds and returns the connector that was connected to **encoder** as specified by the **state**.

If there is no connector in **state** which previously had **encoder** connected to it, this function will return NULL. While this may seem like an invalid use case, it is sometimes useful to differentiate commits which had no prior connectors attached to **encoder** vs ones that did (and to inspect their state). This is especially true in enable hooks because the pipeline has changed.

Return

The old connector connected to **encoder**, or NULL if the encoder is not connected.


```
struct drm_connector *drm_atomic_get_new_connector_for_encoder(const struct  
                                                             drm_atomic_state  
                                                             *state, struct  
                                                             drm_encoder  
                                                             *encoder)
```

Get new connector for an encoder

Parameters

const struct *drm_atomic_state* *state
Atomic state

struct *drm_encoder* *encoder
The encoder to fetch the connector state for

Description

This function finds and returns the connector that will be connected to **encoder** as specified by the **state**.

If there is no connector in **state** which will have **encoder** connected to it, this function will return NULL. While this may seem like an invalid use case, it is sometimes useful to differentiate commits which have no connectors attached to **encoder** vs ones that do (and to inspect their state). This is especially true in disable hooks because the pipeline will change.

Return

The new connector connected to **encoder**, or NULL if the encoder is not connected.

```
struct drm_crtc *drm_atomic_get_old_crtc_for_encoder(struct drm_atomic_state *state,  
                                                    struct drm_encoder *encoder)
```

Get old crtc for an encoder

Parameters

struct *drm_atomic_state* *state
Atomic state

struct *drm_encoder* *encoder
The encoder to fetch the crtc state for

Description

This function finds and returns the crtc that was connected to **encoder** as specified by the **state**.

Return

The old crtc connected to **encoder**, or NULL if the encoder is not connected.

```
struct drm_crtc *drm_atomic_get_new_crtc_for_encoder(struct drm_atomic_state *state,  
                                                    struct drm_encoder *encoder)
```

Get new crtc for an encoder

Parameters

struct *drm_atomic_state* *state
Atomic state

struct drm_encoder *encoder

The encoder to fetch the crtc state for

Description

This function finds and returns the crtc that will be connected to **encoder** as specified by the **state**.

Return

The new crtc connected to **encoder**, or NULL if the encoder is not connected.

struct *drm_connector_state* ***drm_atomic_get_connector_state**(struct *drm_atomic_state* *state, struct *drm_connector* *connector)

get connector state

Parameters

struct drm_atomic_state *state

global atomic state object

struct drm_connector *connector

connector to get state object for

Description

This function returns the connector state for the given connector, allocating it if needed. It will also grab the relevant connector lock to make sure that the state is consistent.

Either the allocated state or the error code encoded into the pointer. When the error is EDEADLK then the w/w mutex code has detected a deadlock and the entire atomic sequence must be restarted. All other errors are fatal.

Return

struct *drm_bridge_state* ***drm_atomic_get_bridge_state**(struct *drm_atomic_state* *state, struct *drm_bridge* *bridge)

get bridge state

Parameters

struct drm_atomic_state *state

global atomic state object

struct drm_bridge *bridge

bridge to get state object for

Description

This function returns the bridge state for the given bridge, allocating it if needed. It will also grab the relevant bridge lock to make sure that the state is consistent.

Either the allocated state or the error code encoded into the pointer. When the error is EDEADLK then the w/w mutex code has detected a deadlock and the entire atomic sequence must be restarted.

Return

struct *drm_bridge_state* ***drm_atomic_get_old_bridge_state**(const struct *drm_atomic_state* *state, struct *drm_bridge* *bridge)

get old bridge state, if it exists

Parameters

const struct drm_atomic_state *state

global atomic state object

struct drm_bridge *bridge

bridge to grab

Description

This function returns the old bridge state for the given bridge, or NULL if the bridge is not part of the global atomic state.

```
struct drm_bridge_state *drm_atomic_get_new_bridge_state(const struct drm_atomic_state
                                                         *state, struct drm_bridge
                                                         *bridge)
```

get new bridge state, if it exists

Parameters

const struct drm_atomic_state *state

global atomic state object

struct drm_bridge *bridge

bridge to grab

Description

This function returns the new bridge state for the given bridge, or NULL if the bridge is not part of the global atomic state.

```
int drm_atomic_add_encoder_bridges(struct drm_atomic_state *state, struct drm_encoder
                                   *encoder)
```

add bridges attached to an encoder

Parameters

struct drm_atomic_state *state

atomic state

struct drm_encoder *encoder

DRM encoder

Description

This function adds all bridges attached to **encoder**. This is needed to add bridge states to **state** and make them available when *drm_bridge_funcs.atomic_check()*, *drm_bridge_funcs.atomic_pre_enable()*, *drm_bridge_funcs.atomic_enable()*, *drm_bridge_funcs.atomic_disable_post_disable()* are called.

Return

0 on success or can fail with -EDEADLK or -ENOMEM. When the error is EDEADLK then the w/w mutex code has detected a deadlock and the entire atomic sequence must be restarted. All other errors are fatal.

```
int drm_atomic_add_affected_connectors(struct drm_atomic_state *state, struct drm_crtc
                                     *crtc)
```

add connectors for CRTC

Parameters

```
struct drm_atomic_state *state
    atomic state
```

```
struct drm_crtc *crtc
    DRM CRTC
```

Description

This function walks the current configuration and adds all connectors currently using **crtc** to the atomic configuration **state**. Note that this function must acquire the connection mutex. This can potentially cause unneeded serialization if the update is just for the planes on one CRTC. Hence drivers and helpers should only call this when really needed (e.g. when a full modeset needs to happen due to some change).

Return

0 on success or can fail with -EDEADLK or -ENOMEM. When the error is EDEADLK then the w/w mutex code has detected a deadlock and the entire atomic sequence must be restarted. All other errors are fatal.

```
int drm_atomic_add_affected_planes(struct drm_atomic_state *state, struct drm_crtc *crtc)
    add planes for CRTC
```

Parameters

```
struct drm_atomic_state *state
    atomic state
```

```
struct drm_crtc *crtc
    DRM CRTC
```

Description

This function walks the current configuration and adds all planes currently used by **crtc** to the atomic configuration **state**. This is useful when an atomic commit also needs to check all currently enabled plane on **crtc**, e.g. when changing the mode. It's also useful when re-enabling a CRTC to avoid special code to force-enable all planes.

Since acquiring a plane state will always also acquire the w/w mutex of the current CRTC for that plane (if there is any) adding all the plane states for a CRTC will not reduce parallelism of atomic updates.

Return

0 on success or can fail with -EDEADLK or -ENOMEM. When the error is EDEADLK then the w/w mutex code has detected a deadlock and the entire atomic sequence must be restarted. All other errors are fatal.

```
int drm_atomic_check_only(struct drm_atomic_state *state)
    check whether a given config would work
```

Parameters

struct drm_atomic_state *state
atomic configuration to check

Description

Note that this function can return -EDEADLK if the driver needed to acquire more locks but encountered a deadlock. The caller must then do the usual w/w backoff dance and restart. All other errors are fatal.

Return

0 on success, negative error code on failure.

int **drm_atomic_commit**(struct *drm_atomic_state* *state)
commit configuration atomically

Parameters

struct drm_atomic_state *state
atomic configuration to check

Description

Note that this function can return -EDEADLK if the driver needed to acquire more locks but encountered a deadlock. The caller must then do the usual w/w backoff dance and restart. All other errors are fatal.

This function will take its own reference on **state**. Callers should always release their reference with *drm_atomic_state_put()*.

Return

0 on success, negative error code on failure.

int **drm_atomic_nonblocking_commit**(struct *drm_atomic_state* *state)
atomic nonblocking commit

Parameters

struct drm_atomic_state *state
atomic configuration to check

Description

Note that this function can return -EDEADLK if the driver needed to acquire more locks but encountered a deadlock. The caller must then do the usual w/w backoff dance and restart. All other errors are fatal.

This function will take its own reference on **state**. Callers should always release their reference with *drm_atomic_state_put()*.

Return

0 on success, negative error code on failure.

void **drm_atomic_print_new_state**(const struct *drm_atomic_state* *state, struct *drm_printer* *p)
prints drm atomic state

Parameters

const struct drm_atomic_state *state

atomic configuration to check

struct drm_printer *p

drm printer

Description

This functions prints the drm atomic state snapshot using the drm printer which is passed to it. This snapshot can be used for debugging purposes.

Note that this function looks into the new state objects and hence its not safe to be used after the call to *drm_atomic_helper_commit_hw_done()*.

void **drm_state_dump**(struct *drm_device* *dev, struct *drm_printer* *p)

dump entire device atomic state

Parameters

struct drm_device *dev

the drm device

struct drm_printer *p

where to print the state to

Description

Just for debugging. Drivers might want an option to dump state to dmesg in case of error irq's. (Hint, you probably want to ratelimit this!)

The caller must wrap this *drm_modeset_lock_all_ctx()* and *drm_modeset_drop_locks()*. If this is called from error irq handler, it should not be enabled by default - if you are debugging errors you might not care that this is racey, but calling this without all modeset locks held is inherently unsafe.

4.4.3 Atomic Mode Setting IOCTL and UAPI Functions

This file contains the marshalling and demarshalling glue for the atomic UAPI in all its forms: The monster ATOMIC IOCTL itself, code for GET_PROPERTY and SET_PROPERTY IOCTLs. Plus interface functions for compatibility helpers and drivers which have special needs to construct their own atomic updates, e.g. for load detect or similar.

int **drm_atomic_set_mode_for_crtc**(struct *drm_crtc_state* *state, const struct *drm_display_mode* *mode)

set mode for CRTC

Parameters

struct drm_crtc_state *state

the CRTC whose incoming state to update

const struct drm_display_mode *mode

kernel-internal mode to use for the CRTC, or NULL to disable

Description

Set a mode (originating from the kernel) on the desired CRTC state and update the enable property.

Return

Zero on success, error code on failure. Cannot return -EDEADLK.

int **drm_atomic_set_mode_prop_for_crtc**(struct *drm_crtc_state* *state, struct *drm_property_blob* *blob)

set mode for CRTC

Parameters

struct drm_crtc_state *state

the CRTC whose incoming state to update

struct drm_property_blob *blob

pointer to blob property to use for mode

Description

Set a mode (originating from a blob property) on the desired CRTC state. This function will take a reference on the blob property for the CRTC state, and release the reference held on the state's existing mode property, if any was set.

Return

Zero on success, error code on failure. Cannot return -EDEADLK.

int **drm_atomic_set_crtc_for_plane**(struct *drm_plane_state* *plane_state, struct *drm_crtc* *crtc)

set CRTC for plane

Parameters

struct drm_plane_state *plane_state

the plane whose incoming state to update

struct drm_crtc *crtc

CRTC to use for the plane

Description

Changing the assigned CRTC for a plane requires us to grab the lock and state for the new CRTC, as needed. This function takes care of all these details besides updating the pointer in the state object itself.

Return

0 on success or can fail with -EDEADLK or -ENOMEM. When the error is EDEADLK then the w/w mutex code has detected a deadlock and the entire atomic sequence must be restarted. All other errors are fatal.

void **drm_atomic_set_fb_for_plane**(struct *drm_plane_state* *plane_state, struct *drm_framebuffer* *fb)

set framebuffer for plane

Parameters

struct drm_plane_state *plane_state

atomic state object for the plane

struct drm_framebuffer *fb

fb to use for the plane

Description

Changing the assigned framebuffer for a plane requires us to grab a reference to the new fb and drop the reference to the old fb, if there is one. This function takes care of all these details besides updating the pointer in the state object itself.

```
int drm_atomic_set_crtc_for_connector(struct drm_connector_state *conn_state, struct drm_crtc *crtc)
```

set CRTC for connector

Parameters

struct *drm_connector_state* *conn_state

atomic state object for the connector

struct *drm_crtc* *crtc

CRTC to use for the connector

Description

Changing the assigned CRTC for a connector requires us to grab the lock and state for the new CRTC, as needed. This function takes care of all these details besides updating the pointer in the state object itself.

Return

0 on success or can fail with -EDEADLK or -ENOMEM. When the error is EDEADLK then the w/w mutex code has detected a deadlock and the entire atomic sequence must be restarted. All other errors are fatal.

4.5 CRTC Abstraction

A CRTC represents the overall display pipeline. It receives pixel data from *drm_plane* and blends them together. The *drm_display_mode* is also attached to the CRTC, specifying display timings. On the output side the data is fed to one or more *drm_encoder*, which are then each connected to one *drm_connector*.

To create a CRTC, a KMS drivers allocates and zeroes an instances of *struct drm_crtc* (possibly as part of a larger structure) and registers it with a call to *drm_crtc_init_with_planes()*.

The CRTC is also the entry point for legacy modeset operations, see *drm_crtc_funcs.set_config*, legacy plane operations, see *drm_crtc_funcs.page_flip* and *drm_crtc_funcs.cursor_set2*, and other legacy operations like *drm_crtc_funcs.gamma_set*. For atomic drivers all these features are controlled through *drm_property* and *drm_mode_config_funcs.atomic_check*.

4.5.1 CRTC Functions Reference

struct **drm_crtc_state**
 mutable CRTC state

Definition:

```
struct drm_crtc_state {
    struct drm_crtc *crtc;
    bool enable;
    bool active;
    bool planes_changed : 1;
    bool mode_changed : 1;
    bool active_changed : 1;
    bool connectors_changed : 1;
    bool zpos_changed : 1;
    bool color_mgmt_changed : 1;
    bool no_vblank : 1;
    u32 plane_mask;
    u32 connector_mask;
    u32 encoder_mask;
    struct drm_display_mode adjusted_mode;
    struct drm_display_mode mode;
    struct drm_property_blob *mode_blob;
    struct drm_property_blob *degamma_lut;
    struct drm_property_blob *ctm;
    struct drm_property_blob *gamma_lut;
    u32 target_vblank;
    bool async_flip;
    bool vrr_enabled;
    bool self_refresh_active;
    enum drm_scaling_filter scaling_filter;
    struct drm_pending_vblank_event *event;
    struct drm_crtc_commit *commit;
    struct drm_atomic_state *state;
};
```

Members

crtc

backpointer to the CRTC

enable

Whether the CRTC should be enabled, gates all other state. This controls reservations of shared resources. Actual hardware state is controlled by **active**.

active

Whether the CRTC is actively displaying (used for DPMS). Implies that **enable** is set. The driver must not release any shared resources if **active** is set to false but **enable** still true, because userspace expects that a DPMS ON always succeeds.

Hence drivers must not consult **active** in their various [drm_mode_config_funcs.atomic_check](#) callback to reject an atomic commit. They can consult it to aid in the computation of derived hardware state, since even in the DPMS OFF state the display hardware

should be as much powered down as when the CRTC is completely disabled through setting **enable** to false.

planes_changed

Planes on this crtc are updated. Used by the atomic helpers and drivers to steer the atomic commit control flow.

mode_changed

mode or **enable** has been changed. Used by the atomic helpers and drivers to steer the atomic commit control flow. See also [*drm_atomic_crtc_needs_modeset\(\)*](#).

Drivers are supposed to set this for any CRTC state changes that require a full modeset. They can also reset it to false if e.g. a **mode** change can be done without a full modeset by only changing scaler settings.

active_changed

active has been toggled. Used by the atomic helpers and drivers to steer the atomic commit control flow. See also [*drm_atomic_crtc_needs_modeset\(\)*](#).

connectors_changed

Connectors to this crtc have been updated, either in their state or routing. Used by the atomic helpers and drivers to steer the atomic commit control flow. See also [*drm_atomic_crtc_needs_modeset\(\)*](#).

Drivers are supposed to set this as-needed from their own atomic check code, e.g. from [*drm_encoder_helper_funcs.atomic_check*](#)

zpos_changed

zpos values of planes on this crtc have been updated. Used by the atomic helpers and drivers to steer the atomic commit control flow.

color_mgmt_changed

Color management properties have changed (**gamma_lut**, **degamma_lut** or **ctm**). Used by the atomic helpers and drivers to steer the atomic commit control flow.

no_vblank

Reflects the ability of a CRTC to send VBLANK events. This state usually depends on the pipeline configuration. If set to true, DRM atomic helpers will send out a fake VBLANK event during display updates after all hardware changes have been committed. This is implemented in [*drm_atomic_helper_fake_vblank\(\)*](#).

One usage is for drivers and/or hardware without support for VBLANK interrupts. Such drivers typically do not initialize vblanking (i.e., call [*drm_vblank_init\(\)*](#) with the number of CRTCs). For CRTCs without initialized vblanking, this field is set to true in [*drm_atomic_helper_check_modeset\(\)*](#), and a fake VBLANK event will be send out on each update of the display pipeline by [*drm_atomic_helper_fake_vblank\(\)*](#).

Another usage is CRTCs feeding a writeback connector operating in oneshot mode. In this case the fake VBLANK event is only generated when a job is queued to the writeback connector, and we want the core to fake VBLANK events when this part of the pipeline hasn't changed but others had or when the CRTC and connectors are being disabled.

[*__drm_atomic_helper_crtc_duplicate_state\(\)*](#) will not reset the value from the current state, the CRTC driver is then responsible for updating this field when needed.

Note that the combination of [*drm_crtc_state.event*](#) == NULL and [*drm_crtc_state.no_blank*](#) == true is valid and usually used when the writeback connector attached to

the CRTC has a new job queued. In this case the driver will send the VBLANK event on its own when the writeback job is complete.

plane_mask

Bitmask of `drm_plane_mask(plane)` of planes attached to this CRTC.

connector_mask

Bitmask of `drm_connector_mask(connector)` of connectors attached to this CRTC.

encoder_mask

Bitmask of `drm_encoder_mask(encoder)` of encoders attached to this CRTC.

adjusted_mode

Internal display timings which can be used by the driver to handle differences between the mode requested by userspace in **mode** and what is actually programmed into the hardware.

For drivers using *drm_bridge*, this stores hardware display timings used between the CRTC and the first bridge. For other drivers, the meaning of the `adjusted_mode` field is purely driver implementation defined information, and will usually be used to store the hardware display timings used between the CRTC and encoder blocks.

mode

Display timings requested by userspace. The driver should try to match the refresh rate as close as possible (but note that it's undefined what exactly is close enough, e.g. some of the HDMI modes only differ in less than 1% of the refresh rate). The active width and height as observed by userspace for positioning planes must match exactly.

For external connectors where the sink isn't fixed (like with a built-in panel), this mode here should match the physical mode on the wire to the last details (i.e. including sync polarities and everything).

mode_blob

drm_property_blob for **mode**, for exposing the mode to atomic userspace.

degamma_lut

Lookup table for converting framebuffer pixel data before apply the color conversion matrix **ctm**. See *drm_crtc_enable_color_mgmt()*. The blob (if not NULL) is an array of struct `drm_color_lut`.

ctm

Color transformation matrix. See *drm_crtc_enable_color_mgmt()*. The blob (if not NULL) is a struct `drm_color_ctm`.

gamma_lut

Lookup table for converting pixel data after the color conversion matrix **ctm**. See *drm_crtc_enable_color_mgmt()*. The blob (if not NULL) is an array of struct `drm_color_lut`.

Note that for mostly historical reasons stemming from Xorg heritage, this is also used to store the color map (also sometimes color lut, CLUT or color palette) for indexed formats like `DRM_FORMAT_C8`.

target_vblank

Target vertical blank period when a page flip should take effect.

async_flip

This is set when `DRM_MODE_PAGE_FLIP_ASYNC` is set in the legacy `PAGE_FLIP_IOCTL`.

It's not wired up for the atomic IOCTL itself yet.

vrp_enabled

Indicates if variable refresh rate should be enabled for the CRTC. Support for the requested vrr state will depend on driver and hardware capability - lacking support is not treated as failure.

self_refresh_active

Used by the self refresh helpers to denote when a self refresh transition is occurring. This will be set on enable/disable callbacks when self refresh is being enabled or disabled. In some cases, it may not be desirable to fully shut off the crtc during self refresh. CRTC's can inspect this flag and determine the best course of action.

scaling_filter

Scaling filter to be applied

event

Optional pointer to a DRM event to signal upon completion of the state update. The driver must send out the event when the atomic commit operation completes. There are two cases:

- The event is for a CRTC which is being disabled through this atomic commit. In that case the event can be sent out any time after the hardware has stopped scanning out the current framebuffers. It should contain the timestamp and counter for the last vblank before the display pipeline was shut off. The simplest way to achieve that is calling `drm_crtc_send_vblank_event()` somewhere after `drm_crtc_vblank_off()` has been called.
- For a CRTC which is enabled at the end of the commit (even when it undergoes a full modeset) the vblank timestamp and counter must be for the vblank right before the first frame that scans out the new set of buffers. Again the event can only be sent out after the hardware has stopped scanning out the old buffers.
- Events for disabled CRTCs are not allowed, and drivers can ignore that case.

For very simple hardware without VBLANK interrupt, enabling `struct drm_crtc_state.no_vblank` makes DRM's atomic commit helpers send a fake VBLANK event at the end of the display update after all hardware changes have been applied. See `drm_atomic_helper_fake_vblank()`.

For more complex hardware this can be handled by the `drm_crtc_send_vblank_event()` function, which the driver should call on the provided event upon completion of the atomic commit. Note that if the driver supports vblank signalling and timestamping the vblank counters and timestamps must agree with the ones returned from page flip events. With the current vblank helper infrastructure this can be achieved by holding a vblank reference while the page flip is pending, acquired through `drm_crtc_vblank_get()` and released with `drm_crtc_vblank_put()`. Drivers are free to implement their own vblank counter and timestamp tracking though, e.g. if they have accurate timestamp registers in hardware.

For hardware which supports some means to synchronize vblank interrupt delivery with committing display state there's also `drm_crtc_arm_vblank_event()`. See the documentation of that function for a detailed discussion of the constraints it needs to be used safely.

If the device can't notify of flip completion in a race-free way at all, then the event should be armed just after the page flip is committed. In the worst case the driver will send the event to userspace one frame too late. This doesn't allow for a real atomic update, but it should avoid tearing.

commit

This tracks how the commit for this update proceeds through the various phases. This is never cleared, except when we destroy the state, so that subsequent commits can synchronize with previous ones.

state

backpointer to global `drm_atomic_state`

Description

Note that the distinction between **enable** and **active** is rather subtle: Flipping **active** while **enable** is set without changing anything else may never return in a failure from the `drm_mode_config_funcs.atomic_check` callback. Userspace assumes that a DPMS On will always succeed. In other words: **enable** controls resource assignment, **active** controls the actual hardware state.

The three booleans `active_changed`, `connectors_changed` and `mode_changed` are intended to indicate whether a full modeset is needed, rather than strictly describing what has changed in a commit. See also: `drm_atomic_crtc_needs_modeset()`

struct `drm_crtc_funcs`

control CRTC's for a given device

Definition:

```
struct drm_crtc_funcs {
    void (*reset)(struct drm_crtc *crtc);
    int (*cursor_set)(struct drm_crtc *crtc, struct drm_file *file_priv,
    ↪ uint32_t handle, uint32_t width, uint32_t height);
    int (*cursor_set2)(struct drm_crtc *crtc, struct drm_file *file_priv,
    ↪ uint32_t handle, uint32_t width, uint32_t height, int32_t hot_x, int32_t hot_
    ↪ y);
    int (*cursor_move)(struct drm_crtc *crtc, int x, int y);
    int (*gamma_set)(struct drm_crtc *crtc, u16 *r, u16 *g, u16 *b, uint32_t
    ↪ size, struct drm_modeset_acquire_ctx *ctx);
    void (*destroy)(struct drm_crtc *crtc);
    int (*set_config)(struct drm_mode_set *set, struct drm_modeset_acquire_ctx
    ↪ *ctx);
    int (*page_flip)(struct drm_crtc *crtc, struct drm_framebuffer *fb, struct
    ↪ drm_pending_vblank_event *event, uint32_t flags, struct drm_modeset_acquire_
    ↪ ctx *ctx);
    int (*page_flip_target)(struct drm_crtc *crtc, struct drm_framebuffer *fb,
    ↪ struct drm_pending_vblank_event *event, uint32_t flags, uint32_t target,
    ↪ struct drm_modeset_acquire_ctx *ctx);
    int (*set_property)(struct drm_crtc *crtc, struct drm_property *property,
    ↪ uint64_t val);
    struct drm_crtc_state *(*atomic_duplicate_state)(struct drm_crtc *crtc);
    void (*atomic_destroy_state)(struct drm_crtc *crtc, struct drm_crtc_state
    ↪ *state);
    int (*atomic_set_property)(struct drm_crtc *crtc, struct drm_crtc_state
    ↪ *state, struct drm_property *property, uint64_t val);
    int (*atomic_get_property)(struct drm_crtc *crtc, const struct drm_crtc_
    ↪ state *state, struct drm_property *property, uint64_t *val);
    int (*late_register)(struct drm_crtc *crtc);
}
```

```
void (*early_unregister)(struct drm_crtc *crtc);
int (*set_crc_source)(struct drm_crtc *crtc, const char *source);
int (*verify_crc_source)(struct drm_crtc *crtc, const char *source, size_t_
↪ *values_cnt);
const char *const *(*get_crc_sources)(struct drm_crtc *crtc, size_t_
↪ *count);
void (*atomic_print_state)(struct drm_printer *p, const struct drm_crtc_
↪ state *state);
u32 (*get_vblank_counter)(struct drm_crtc *crtc);
int (*enable_vblank)(struct drm_crtc *crtc);
void (*disable_vblank)(struct drm_crtc *crtc);
bool (*get_vblank_timestamp)(struct drm_crtc *crtc, int *max_error, ktime_t_
↪ *vblank_time, bool in_vblank_irq);
};
```

Members

reset

Reset CRTC hardware and software state to off. This function isn't called by the core directly, only through [drm_mode_config_reset\(\)](#). It's not a helper hook only for historical reasons.

Atomic drivers can use [drm_atomic_helper_crtc_reset\(\)](#) to reset atomic state using this hook.

cursor_set

Update the cursor image. The cursor position is relative to the CRTC and can be partially or fully outside of the visible area.

Note that contrary to all other KMS functions the legacy cursor entry points don't take a framebuffer object, but instead take directly a raw buffer object id from the driver's buffer manager (which is either GEM or TTM for current drivers).

This entry point is deprecated, drivers should instead implement universal plane support and register a proper cursor plane using [drm_crtc_init_with_planes\(\)](#).

This callback is optional

RETURNS:

0 on success or a negative error code on failure.

cursor_set2

Update the cursor image, including hotspot information. The hotspot must not affect the cursor position in CRTC coordinates, but is only meant as a hint for virtualized display hardware to coordinate the guests and hosts cursor position. The cursor hotspot is relative to the cursor image. Otherwise this works exactly like **cursor_set**.

This entry point is deprecated, drivers should instead implement universal plane support and register a proper cursor plane using [drm_crtc_init_with_planes\(\)](#).

This callback is optional.

RETURNS:

0 on success or a negative error code on failure.

cursor_move

Update the cursor position. The cursor does not need to be visible when this hook is called.

This entry point is deprecated, drivers should instead implement universal plane support and register a proper cursor plane using `drm_crtc_init_with_planes()`.

This callback is optional.

RETURNS:

0 on success or a negative error code on failure.

gamma_set

Set gamma on the CRTC.

This callback is optional.

Atomic drivers who want to support gamma tables should implement the atomic color management support, enabled by calling `drm_crtc_enable_color_mgmt()`, which then supports the legacy gamma interface through the `drm_atomic_helper_legacy_gamma_set()` compatibility implementation.

destroy

Clean up CRTC resources. This is only called at driver unload time through `drm_mode_config_cleanup()` since a CRTC cannot be hotplugged in DRM.

set_config

This is the main legacy entry point to change the modeset state on a CRTC. All the details of the desired configuration are passed in a `struct drm_mode_set` - see there for details.

Drivers implementing atomic modeset should use `drm_atomic_helper_set_config()` to implement this hook.

RETURNS:

0 on success or a negative error code on failure.

page_flip

Legacy entry point to schedule a flip to the given framebuffer.

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 (except when requested otherwise through the `DRM_MODE_PAGE_FLIP_ASYNC` flag). 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 this hook with a pointer to the new frame buffer.

The driver must wait for any pending rendering to the new framebuffer to complete before executing the flip. It should also wait for any pending rendering from other drivers if the underlying buffer is a shared dma-buf.

An application can request to be notified when the page flip has completed. The drm core will supply a `struct drm_event` in the event parameter in this case. This can be handled by the `drm_crtc_send_vblank_event()` function, which the driver should call on the provided event upon completion of the flip. Note that if the driver supports vblank signalling and timestamping the vblank counters and timestamps must agree with the ones returned from page flip events. With the current vblank helper infrastructure this can be achieved by holding a vblank reference while the page flip is pending, acquired through `drm_crtc_vblank_get()` and released with `drm_crtc_vblank_put()`. Drivers are free to

implement their own vblank counter and timestamp tracking though, e.g. if they have accurate timestamp registers in hardware.

This callback is optional.

NOTE:

Very early versions of the KMS ABI mandated that the driver must block (but not reject) any rendering to the old framebuffer until the flip operation has completed and the old framebuffer is no longer visible. This requirement has been lifted, and userspace is instead expected to request delivery of an event and wait with recycling old buffers until such has been received.

RETURNS:

0 on success or a negative error code on failure. Note that if a page flip operation is already pending the callback should return `-EBUSY`. Pageflips on a disabled CRTC (either by setting a NULL mode or just runtime disabled through DPMS respectively the new atomic “ACTIVE” state) should result in an `-EINVAL` error code. Note that `drm_atomic_helper_page_flip()` checks this already for atomic drivers.

page_flip_target

Same as **page_flip** but with an additional parameter specifying the absolute target vertical blank period (as reported by `drm_crtc_vblank_count()`) when the flip should take effect.

Note that the core code calls `drm_crtc_vblank_get` before this entry point, and will call `drm_crtc_vblank_put` if this entry point returns any non-0 error code. It's the driver's responsibility to call `drm_crtc_vblank_put` after this entry point returns 0, typically when the flip completes.

set_property

This is the legacy entry point to update a property attached to the CRTC.

This callback is optional if the driver does not support any legacy driver-private properties. For atomic drivers it is not used because property handling is done entirely in the DRM core.

RETURNS:

0 on success or a negative error code on failure.

atomic_duplicate_state

Duplicate the current atomic state for this CRTC and return it. The core and helpers guarantee that any atomic state duplicated with this hook and still owned by the caller (i.e. not transferred to the driver by calling `drm_mode_config_funcs.atomic_commit`) will be cleaned up by calling the **atomic_destroy_state** hook in this structure.

This callback is mandatory for atomic drivers.

Atomic drivers which don't subclass `struct drm_crtc_state` should use `drm_atomic_helper_crtc_duplicate_state()`. Drivers that subclass the state structure to extend it with driver-private state should use `__drm_atomic_helper_crtc_duplicate_state()` to make sure shared state is duplicated in a consistent fashion across drivers.

It is an error to call this hook before `drm_crtc.state` has been initialized correctly.

NOTE:

If the duplicate state references refcounted resources this hook must acquire a reference for each of them. The driver must release these references again in **atomic_destroy_state**.

RETURNS:

Duplicated atomic state or NULL when the allocation failed.

atomic_destroy_state

Destroy a state duplicated with **atomic_duplicate_state** and release or unreference all resources it references

This callback is mandatory for atomic drivers.

atomic_set_property

Decode a driver-private property value and store the decoded value into the passed-in state structure. Since the atomic core decodes all standardized properties (even for extensions beyond the core set of properties which might not be implemented by all drivers) this requires drivers to subclass the state structure.

Such driver-private properties should really only be implemented for truly hardware/vendor specific state. Instead it is preferred to standardize atomic extension and decode the properties used to expose such an extension in the core.

Do not call this function directly, use `drm_atomic_crtc_set_property()` instead.

This callback is optional if the driver does not support any driver-private atomic properties.

NOTE:

This function is called in the state assembly phase of atomic modesets, which can be aborted for any reason (including on userspace's request to just check whether a configuration would be possible). Drivers **MUST NOT** touch any persistent state (hardware or software) or data structures except the passed in **state** parameter.

Also since userspace controls in which order properties are set this function must not do any input validation (since the state update is incomplete and hence likely inconsistent). Instead any such input validation must be done in the various `atomic_check` callbacks.

RETURNS:

0 if the property has been found, `-EINVAL` if the property isn't implemented by the driver (which should never happen, the core only asks for properties attached to this CRTC). No other validation is allowed by the driver. The core already checks that the property value is within the range (integer, valid enum value, ...) the driver set when registering the property.

atomic_get_property

Reads out the decoded driver-private property. This is used to implement the `GETCRTC_IOCTL`.

Do not call this function directly, use `drm_atomic_crtc_get_property()` instead.

This callback is optional if the driver does not support any driver-private atomic properties.

RETURNS:

0 on success, `-EINVAL` if the property isn't implemented by the driver (which should never happen, the core only asks for properties attached to this CRTC).

late_register

This optional hook can be used to register additional userspace interfaces attached to the crtc like debugfs interfaces. It is called late in the driver load sequence from [drm_dev_register\(\)](#). Everything added from this callback should be unregistered in the [early_unregister](#) callback.

Returns:

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

early_unregister

This optional hook should be used to unregister the additional userspace interfaces attached to the crtc from **late_register**. It is called from [drm_dev_unregister\(\)](#), early in the driver unload sequence to disable userspace access before data structures are torn-down.

set_crc_source

Changes the source of CRC checksums of frames at the request of userspace, typically for testing purposes. The sources available are specific of each driver and a NULL value indicates that CRC generation is to be switched off.

When CRC generation is enabled, the driver should call [drm_crtc_add_crc_entry\(\)](#) at each frame, providing any information that characterizes the frame contents in the `crcN` arguments, as provided from the configured source. Drivers must accept an “auto” source name that will select a default source for this CRTC.

This may trigger an atomic modeset commit if necessary, to enable CRC generation.

Note that “auto” can depend upon the current modeset configuration, e.g. it could pick an encoder or output specific CRC sampling point.

This callback is optional if the driver does not support any CRC generation functionality.

RETURNS:

0 on success or a negative error code on failure.

verify_crc_source

verifies the source of CRC checksums of frames before setting the source for CRC and during crc open. Source parameter can be NULL while disabling crc source.

This callback is optional if the driver does not support any CRC generation functionality.

RETURNS:

0 on success or a negative error code on failure.

get_crc_sources

Driver callback for getting a list of all the available sources for CRC generation. This callback depends upon [verify_crc_source](#), So [verify_crc_source](#) callback should be implemented before implementing this. Driver can pass full list of available crc sources, this callback does the verification on each crc-source before passing it to userspace.

This callback is optional if the driver does not support exporting of possible CRC sources list.

RETURNS:

a constant character pointer to the list of all the available CRC sources. On failure driver should return NULL. count should be updated with number of sources in list. if zero we

don't process any source from the list.

atomic_print_state

If driver subclasses *struct `drm_crtc_state`*, it should implement this optional hook for printing additional driver specific state.

Do not call this directly, use `drm_atomic_crtc_print_state()` instead.

get_vblank_counter

Driver callback for fetching a raw hardware vblank counter for the CRTC. It's meant to be used by new drivers as the replacement of *`drm_driver.get_vblank_counter`* hook.

This callback is optional. If a device doesn't have a hardware counter, the driver can simply leave the hook as NULL. The DRM core will account for missed vblank events while interrupts where disabled based on system timestamps.

Wraparound handling and loss of events due to modesetting is dealt with in the DRM core code, as long as drivers call *`drm_crtc_vblank_off()`* and *`drm_crtc_vblank_on()`* when disabling or enabling a CRTC.

See also *`drm_device.vblank_disable_immediate`* and *`drm_device.max_vblank_count`*.

Returns:

Raw vblank counter value.

enable_vblank

Enable vblank interrupts for the CRTC. It's meant to be used by new drivers as the replacement of *`drm_driver.enable_vblank`* hook.

Returns:

Zero on success, appropriate `errno` if the vblank interrupt cannot be enabled.

disable_vblank

Disable vblank interrupts for the CRTC. It's meant to be used by new drivers as the replacement of *`drm_driver.disable_vblank`* hook.

get_vblank_timestamp

Called by `drm_get_last_vbltimestamp()`. Should return a precise timestamp when the most recent vblank interval ended or will end.

Specifically, the timestamp in **`vblank_time`** should correspond as closely as possible to the time when the first video scanline of the video frame after the end of vblank will start scanning out, the time immediately after end of the vblank interval. If the **`crtc`** is currently inside vblank, this will be a time in the future. If the **`crtc`** is currently scanning out a frame, this will be the past start time of the current scanout. This is meant to adhere to the OpenML OML_sync_control extension specification.

Parameters:

`crtc`:

CRTC for which timestamp should be returned.

`max_error`:

Maximum allowable timestamp error in nanoseconds. Implementation should strive to provide timestamp with an error of at most `max_error` nanoseconds. Returns true upper bound on error for timestamp.

vblank_time:

Target location for returned vblank timestamp.

in_vblank_irq:

True when called from `drm_crtc_handle_vblank()`. Some drivers need to apply some workarounds for gpu-specific vblank irq quirks if flag is set.

Returns:

True on success, false on failure, which means the core should fallback to a simple timestamp taken in `drm_crtc_handle_vblank()`.

Description

The `drm_crtc_funcs` structure is the central CRTC management structure in the DRM. Each CRTC controls one or more connectors (note that the name CRTC is simply historical, a CRTC may control LVDS, VGA, DVI, TV out, etc. connectors, not just CRTs).

Each driver is responsible for filling out this structure at startup time, in addition to providing other modesetting features, like i2c and DDC bus accessors.

struct `drm_crtc`

central CRTC control structure

Definition:

```
struct drm_crtc {
    struct drm_device *dev;
    struct device_node *port;
    struct list_head head;
    char *name;
    struct drm_modeset_lock mutex;
    struct drm_mode_object base;
    struct drm_plane *primary;
    struct drm_plane *cursor;
    unsigned index;
    int cursor_x;
    int cursor_y;
    bool enabled;
    struct drm_display_mode mode;
    struct drm_display_mode hwmode;
    int x;
    int y;
    const struct drm_crtc_funcs *funcs;
    uint32_t gamma_size;
    uint16_t *gamma_store;
    const struct drm_crtc_helper_funcs *helper_private;
    struct drm_object_properties properties;
    struct drm_property *scaling_filter_property;
    struct drm_crtc_state *state;
    struct list_head commit_list;
    spinlock_t commit_lock;
    struct dentry *debugfs_entry;
    struct drm_crtc_crc crc;
    unsigned int fence_context;
```

```

spinlock_t fence_lock;
unsigned long fence_seqno;
char timeline_name[32];
struct drm_self_refresh_data *self_refresh_data;
};

```

Members

- dev**
parent DRM device
- port**
OF node used by [drm_of_find_possible_crtcs\(\)](#).
- head**
List of all CRTC's on **dev**, linked from [drm_mode_config.crtc_list](#). Invariant over the lifetime of **dev** and therefore does not need locking.
- name**
human readable name, can be overwritten by the driver
- mutex**
This provides a read lock for the overall CRTC state (mode, dpms state, ...) and a write lock for everything which can be update without a full modeset (fb, cursor data, CRTC properties ...). A full modeset also need to grab [drm_mode_config.connection_mutex](#).
For atomic drivers specifically this protects **state**.
- base**
base KMS object for ID tracking etc.
- primary**
Primary plane for this CRTC. Note that this is only relevant for legacy IOCTL, it specifies the plane implicitly used by the SETCRTC and PAGE_FLIP IOCTLs. It does not have any significance beyond that.
- cursor**
Cursor plane for this CRTC. Note that this is only relevant for legacy IOCTL, it specifies the plane implicitly used by the SETCURSOR and SETCURSOR2 IOCTLs. It does not have any significance beyond that.
- index**
Position inside the mode_config.list, can be used as an array index. It is invariant over the lifetime of the CRTC.
- cursor_x**
Current x position of the cursor, used for universal cursor planes because the SETCURSOR IOCTL only can update the framebuffer without supplying the coordinates. Drivers should not use this directly, atomic drivers should look at [drm_plane_state.crtc_x](#) of the cursor plane instead.
- cursor_y**
Current y position of the cursor, used for universal cursor planes because the SETCURSOR IOCTL only can update the framebuffer without supplying the coordinates. Drivers should not use this directly, atomic drivers should look at [drm_plane_state.crtc_y](#) of the cursor plane instead.

enabled

Is this CRTC enabled? Should only be used by legacy drivers, atomic drivers should instead consult `drm_crtc_state.enable` and `drm_crtc_state.active`. Atomic drivers can update this by calling `drm_atomic_helper_update_legacy_modeset_state()`.

mode

Current mode timings. Should only be used by legacy drivers, atomic drivers should instead consult `drm_crtc_state.mode`. Atomic drivers can update this by calling `drm_atomic_helper_update_legacy_modeset_state()`.

hwmode

Programmed mode in hw, after adjustments for encoders, crtc, panel scaling etc. Should only be used by legacy drivers, for high precision vblank timestamps in `drm_crtc_vblank_helper_get_vblank_timestamp()`.

Note that atomic drivers should not use this, but instead use `drm_crtc_state.adjusted_mode`. And for high-precision timestamps `drm_crtc_vblank_helper_get_vblank_timestamp()` used `drm_vblank_crtc.hwmode`, which is filled out by calling `drm_calc_timestamping_constants()`.

x

x position on screen. Should only be used by legacy drivers, atomic drivers should look at `drm_plane_state.crtc_x` of the primary plane instead. Updated by calling `drm_atomic_helper_update_legacy_modeset_state()`.

y

y position on screen. Should only be used by legacy drivers, atomic drivers should look at `drm_plane_state.crtc_y` of the primary plane instead. Updated by calling `drm_atomic_helper_update_legacy_modeset_state()`.

funcs

CRTC control functions

gamma_size

Size of legacy gamma ramp reported to userspace. Set up by calling `drm_mode_crtc_set_gamma_size()`.

Note that atomic drivers need to instead use `drm_crtc_state.gamma_lut`. See `drm_crtc_enable_color_mgmt()`.

gamma_store

Gamma ramp values used by the legacy SETGAMMA and GETGAMMA IOCTLs. Set up by calling `drm_mode_crtc_set_gamma_size()`.

Note that atomic drivers need to instead use `drm_crtc_state.gamma_lut`. See `drm_crtc_enable_color_mgmt()`.

helper_private

mid-layer private data

properties

property tracking for this CRTC

scaling_filter_property

property to apply a particular filter while scaling.

state

Current atomic state for this CRTC.

This is protected by **mutex**. Note that nonblocking atomic commits access the current CRTC state without taking locks. Either by going through the *struct drm_atomic_state* pointers, see *for_each_oldnew_crtc_in_state()*, *for_each_old_crtc_in_state()* and *for_each_new_crtc_in_state()*. Or through careful ordering of atomic commit operations as implemented in the atomic helpers, see *struct drm_crtc_commit*.

commit_list

List of *drm_crtc_commit* structures tracking pending commits. Protected by **commit_lock**. This list holds its own full reference, as does the ongoing commit.

“Note that the commit for a state change is also tracked in *drm_crtc_state.commit*. For accessing the immediately preceding commit in an atomic update it is recommended to just use that pointer in the old CRTC state, since accessing that doesn’t need any locking or list-walking. **commit_list** should only be used to stall for framebuffer cleanup that’s signalled through *drm_crtc_commit.cleanup_done*.”

commit_lock

Spinlock to protect **commit_list**.

debugfs_entry

Debugfs directory for this CRTC.

crc

Configuration settings of CRC capture.

fence_context

timeline context used for fence operations.

fence_lock

spinlock to protect the fences in the *fence_context*.

fence_seqno

Seqno variable used as monotonic counter for the fences created on the CRTC’s timeline.

timeline_name

The name of the CRTC’s fence timeline.

self_refresh_data

Holds the state for the self refresh helpers

Initialized via *drm_self_refresh_helper_init()*.

Description

Each CRTC may have one or more connectors associated with it. This structure allows the CRTC to be controlled.

struct **drm_mode_set**

new values for a CRTC config change

Definition:

```
struct drm_mode_set {
    struct drm_framebuffer *fb;
    struct drm_crtc *crtc;
    struct drm_display_mode *mode;
    uint32_t x;
    uint32_t y;
    struct drm_connector **connectors;
```

```
size_t num_connectors;
};
```

Members

fb

framebuffer to use for new config

crtc

CRTC whose configuration we're about to change

mode

mode timings to use

x

position of this CRTC relative to **fb**

y

position of this CRTC relative to **fb**

connectors

array of connectors to drive with this CRTC if possible

num_connectors

size of **connectors** array

Description

This represents a modeset configuration for the legacy SETCRTC ioctl and is also used internally. Atomic drivers instead use [drm_atomic_state](#).

drm_crtc_alloc_with_planes

`drm_crtc_alloc_with_planes (dev, type, member, primary, cursor, funcs, name, ...)`

Allocate and initialize a new CRTC object with specified primary and cursor planes.

Parameters

dev

DRM device

type

the type of the struct which contains struct [drm_crtc](#)

member

the name of the [drm_crtc](#) within **type**.

primary

Primary plane for CRTC

cursor

Cursor plane for CRTC

funcs

callbacks for the new CRTC

name

printf style format string for the CRTC name, or NULL for default name

...
variable arguments

Description

Allocates and initializes a new `crtc` object. Cleanup is automatically handled through registering `drm_crtc_cleanup()` with `drm_add_action()`.

The **`drm_crtc_funcs.destroy`** hook must be NULL.

Return

Pointer to new `crtc`, or `ERR_PTR` on failure.

unsigned int **`drm_crtc_index`**(const struct *drm_crtc* *crtc)
find the index of a registered CRTC

Parameters

`const struct drm_crtc *crtc`
CRTC to find index for

Description

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

uint32_t **`drm_crtc_mask`**(const struct *drm_crtc* *crtc)
find the mask of a registered CRTC

Parameters

`const struct drm_crtc *crtc`
CRTC to find mask for

Description

Given a registered CRTC, return the mask bit of that CRTC for the *drm_encoder.possible_crtcs* and *drm_plane.possible_crtcs* fields.

struct *drm_crtc* ***`drm_crtc_find`**(struct *drm_device* *dev, struct *drm_file* *file_priv, uint32_t id)
look up a CRTC object from its ID

Parameters

`struct drm_device *dev`
DRM device

`struct drm_file *file_priv`
drm file to check for lease against.

uint32_t **`id`**
drm_mode_object ID

Description

This can be used to look up a CRTC from its userspace ID. Only used by drivers for legacy IOCTLs and interface, nowadays extensions to the KMS userspace interface should be done using *drm_property*.

drm_for_each_crtc

drm_for_each_crtc (crtc, dev)
iterate over all CRTCs

Parameters

crtc
a *struct drm_crtc* as the loop cursor

dev
the *struct drm_device*

Description

Iterate over all CRTCs of **dev**.

drm_for_each_crtc_reverse

drm_for_each_crtc_reverse (crtc, dev)
iterate over all CRTCs in reverse order

Parameters

crtc
a *struct drm_crtc* as the loop cursor

dev
the *struct drm_device*

Description

Iterate over all CRTCs of **dev**.

struct drm_crtc *drm_crtc_from_index(struct *drm_device* *dev, int idx)
find the registered CRTC at an index

Parameters

struct drm_device *dev
DRM device

int idx
index of registered CRTC to find for

Description

Given a CRTC index, return the registered CRTC from DRM device's list of CRTCs with matching index. This is the inverse of *drm_crtc_index()*. It's useful in the vblank callbacks (like *drm_driver.enable_vblank* or *drm_driver.disable_vblank*), since that still deals with indices instead of pointers to *struct drm_crtc*."

int drm_crtc_init_with_planes(struct *drm_device* *dev, struct *drm_crtc* *crtc, struct *drm_plane* *primary, struct *drm_plane* *cursor, const struct *drm_crtc_funcs* *funcs, const char *name, ...)

Initialise a new CRTC object with specified primary and cursor planes.

Parameters

```
struct drm_device *dev
    DRM device

struct drm_crtc *crtc
    CRTC object to init

struct drm_plane *primary
    Primary plane for CRTC

struct drm_plane *cursor
    Cursor plane for CRTC

const struct drm_crtc_funcs *funcs
    callbacks for the new CRTC

const char *name
    printf style format string for the CRTC name, or NULL for default name

...
    variable arguments
```

Description

Initialises a new object created as base part of a driver crtc object. Drivers should use this function instead of `drm_crtc_init()`, which is only provided for backwards compatibility with drivers which do not yet support universal planes). For really simple hardware which has only 1 plane look at `drm_simple_display_pipe_init()` instead. The `drm_crtc_funcs.destroy` hook should call `drm_crtc_cleanup()` and `kfree()` the crtc structure. The crtc structure should not be allocated with `devm_kzalloc()`.

The **primary** and **cursor** planes are only relevant for legacy uAPI, see `drm_crtc.primary` and `drm_crtc.cursor`.

Note

consider using `drm_crtc_alloc_with_planes()` or `drm_crtc_init_with_planes()` instead of `drm_crtc_init_with_planes()` to let the DRM managed resource infrastructure take care of cleanup and deallocation.

Return

Zero on success, error code on failure.

```
int drm_crtc_init_with_planes(struct drm_device *dev, struct drm_crtc *crtc, struct
                             drm_plane *primary, struct drm_plane *cursor, const struct
                             drm_crtc_funcs *funcs, const char *name, ...)
```

Initialise a new CRTC object with specified primary and cursor planes.

Parameters

```
struct drm_device *dev
    DRM device

struct drm_crtc *crtc
    CRTC object to init

struct drm_plane *primary
    Primary plane for CRTC
```

struct drm_plane *cursor

Cursor plane for CRTC

const struct drm_crtc_funcs *funcs

callbacks for the new CRTC

const char *name

printf style format string for the CRTC name, or NULL for default name

...

variable arguments

Description

Initiates a new object created as base part of a driver crtc object. Drivers should use this function instead of *drm_crtc_init()*, which is only provided for backwards compatibility with drivers which do not yet support universal planes). For really simple hardware which has only 1 plane look at *drm_simple_display_pipe_init()* instead.

Cleanup is automatically handled through registering *drm_crtc_cleanup()* with *drm_add_action()*. The crtc structure should be allocated with *drm_kzalloc()*.

The **drm_crtc_funcs.destroy** hook must be NULL.

The **primary** and **cursor** planes are only relevant for legacy uAPI, see *drm_crtc.primary* and *drm_crtc.cursor*.

Return

Zero on success, error code on failure.

void **drm_crtc_cleanup**(struct *drm_crtc* *crtc)

Clean up the core crtc usage

Parameters

struct drm_crtc *crtc

CRTC to cleanup

Description

This function cleans up **crtc** and removes it from the DRM mode setting core. Note that the function does *not* free the crtc structure itself, this is the responsibility of the caller.

int **drm_mode_set_config_internal**(struct *drm_mode_set* *set)

helper to call *drm_mode_config_funcs.set_config*

Parameters

struct drm_mode_set *set

modeset config to set

Description

This is a little helper to wrap internal calls to the *drm_mode_config_funcs.set_config* driver interface. The only thing it adds is correct refcounting dance.

This should only be used by non-atomic legacy drivers.

Return

Zero on success, negative errno on failure.

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

Checks that a framebuffer is big enough for the CRTC viewport

Parameters

const struct *drm_crtc* *crtc
CRTC that framebuffer will be displayed on

int x
x panning

int y
y panning

const struct *drm_display_mode* *mode
mode that framebuffer will be displayed under

const struct *drm_framebuffer* *fb
framebuffer to check size of

```
int drm_crtc_create_scaling_filter_property(struct drm_crtc *crtc, unsigned int  
                                           supported_filters)
```

create a new scaling filter property

Parameters

struct *drm_crtc* *crtc
drm CRTC

unsigned int supported_filters
bitmask of supported scaling filters, must include BIT(DRM_SCALING_FILTER_DEFAULT).

Description

This function lets driver to enable the scaling filter property on a given CRTC.

Return

Zero for success or -errno

4.5.2 Color Management Functions Reference

```
u64 drm_color_ctm_s31_32_to_qm_n(u64 user_input, u32 m, u32 n)
```

Parameters

u64 user_input
input value

u32 m
number of integer bits, only support m <= 32, include the sign-bit

u32 n
number of fractional bits, only support n <= 32

Description

Convert and clamp S31.32 sign-magnitude to Qm.n (signed 2's complement). The sign-bit BIT(m+n-1) and above are 0 for positive value and 1 for negative the range of value is $[-2^{(m-1)}, 2^{(m-1)} - 2^{-n}]$

For example A Q3.12 format number: - required bit: $3 + 12 = 15$ bits - range: $[-2^2, 2^2 - 2^{-15}]$

NOTE

the m can be zero if all bit_precision are used to present fractional
bits like Q0.32

void **drm_crtc_enable_color_mgmt**(struct *drm_crtc* *crtc, uint degamma_lut_size, bool
has_ctm, uint gamma_lut_size)

enable color management properties

Parameters

struct drm_crtc *crtc
DRM CRTC

uint degamma_lut_size
the size of the degamma lut (before CSC)

bool has_ctm
whether to attach ctm_property for CSC matrix

uint gamma_lut_size
the size of the gamma lut (after CSC)

Description

This function lets the driver enable the color correction properties on a CRTC. This includes 3 degamma, csc and gamma properties that userspace can set and 2 size properties to inform the userspace of the lut sizes. Each of the properties are optional. The gamma and degamma properties are only attached if their size is not 0 and ctm_property is only attached if has_ctm is true.

int **drm_mode_crtc_set_gamma_size**(struct *drm_crtc* *crtc, int gamma_size)
set the gamma table size

Parameters

struct drm_crtc *crtc
CRTC to set the gamma table size for

int gamma_size
size of the gamma table

Description

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

Return

Zero on success, negative errno on failure.

int `drm_plane_create_color_properties`(struct *drm_plane* *plane, u32 supported_encodings, u32 supported_ranges, enum `drm_color_encoding` default_encoding, enum `drm_color_range` default_range)
color encoding related plane properties

Parameters

struct `drm_plane` *plane
plane object

u32 `supported_encodings`
bitfield indicating supported color encodings

u32 `supported_ranges`
bitfield indicating supported color ranges

enum `drm_color_encoding` `default_encoding`
default color encoding

enum `drm_color_range` `default_range`
default color range

Description

Create and attach plane specific COLOR_ENCODING and COLOR_RANGE properties to **plane**. The supported encodings and ranges should be provided in `supported_encodings` and `supported_ranges` bitmasks. Each bit set in the bitmask indicates that its number as enum value is supported.

int `drm_color_lut_check`(const struct *drm_property_blob* *lut, u32 tests)
check validity of lookup table

Parameters

const struct `drm_property_blob` *lut
property blob containing LUT to check

u32 `tests`
bitmask of tests to run

Description

Helper to check whether a userspace-provided lookup table is valid and satisfies hardware requirements. Drivers pass a bitmask indicating which of the tests in *drm_color_lut_tests* should be performed.

Returns 0 on success, -EINVAL on failure.

u32 `drm_color_lut_extract`(u32 user_input, int bit_precision)
clamp and round LUT entries

Parameters

u32 `user_input`
input value

int `bit_precision`
number of bits the hw LUT supports

Description

Extract a degamma/gamma LUT value provided by user (in the form of `drm_color_lut` entries) and round it to the precision supported by the hardware.

int **drm_color_lut_size**(const struct *drm_property_blob* *blob)
calculate the number of entries in the LUT

Parameters

const struct **drm_property_blob** *blob
blob containing the LUT

Return

The number of entries in the color LUT stored in **blob**.

enum **drm_color_lut_tests**
hw-specific LUT tests to perform

Constants

DRM_COLOR_LUT_EQUAL_CHANNELS

Checks whether the entries of a LUT all have equal values for the red, green, and blue channels. Intended for hardware that only accepts a single value per LUT entry and assumes that value applies to all three color components.

DRM_COLOR_LUT_NON DECREASING

Checks whether the entries of a LUT are always flat or increasing (never decreasing).

Description

The *drm_color_lut_check()* function takes a bitmask of the values here to determine which tests to apply to a userspace-provided LUT.

4.6 Frame Buffer Abstraction

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 underlying memory manager for allocating backing storage. When creating a frame buffer applications pass a memory handle (or a list of memory handles for multi-planar formats) through the *struct drm_mode_fb_cmd2* argument. For drivers using GEM as their userspace buffer management interface this would be a GEM handle. Drivers are however free to use their own backing storage object handles, e.g. `vmwgfx` directly exposes special TTM handles to userspace and so expects TTM handles in the create ioctl and not GEM handles.

Framebuffers are tracked with *struct drm_framebuffer*. They are published using *drm_framebuffer_init()* - after calling that function userspace can use and access the framebuffer object. The helper function *drm_helper_mode_fill_fb_struct()* can be used to pre-fill the required metadata fields.

The lifetime of a drm framebuffer is controlled with a reference count, drivers can grab additional references with *drm_framebuffer_get()* and drop them again with *drm_framebuffer_put()*. For driver-private framebuffers for which the last reference is never

dropped (e.g. for the fbdev framebuffer when the struct `struct drm_framebuffer` is embedded into the fbdev helper struct) drivers can manually clean up a framebuffer at module unload time with `drm_framebuffer_unregister_private()`. But doing this is not recommended, and it's better to have a normal free-standing `struct drm_framebuffer`.

4.6.1 Frame Buffer Functions Reference

struct **drm_framebuffer_funcs**
framebuffer hooks

Definition:

```
struct drm_framebuffer_funcs {
    void (*destroy)(struct drm_framebuffer *framebuffer);
    int (*create_handle)(struct drm_framebuffer *fb, struct drm_file *file_priv,
    ↪ unsigned int *handle);
    int (*dirty)(struct drm_framebuffer *framebuffer, struct drm_file *file_
    ↪ priv, unsigned flags, unsigned color, struct drm_clip_rect *clips, unsigned
    ↪ num_clips);
};
```

Members

destroy

Clean up framebuffer resources, specifically also unreference the backing storage. The core guarantees to call this function for every framebuffer successfully created by calling `drm_mode_config_funcs.fb_create`. Drivers must also call `drm_framebuffer_cleanup()` to release DRM core resources for this framebuffer.

create_handle

Create a buffer handle in the driver-specific buffer manager (either GEM or TTM) valid for the passed-in `struct drm_file`. This is used by the core to implement the GETFB IOCTL, which returns (for sufficiently privileged user) also a native buffer handle. This can be used for seamless transitions between modesetting clients by copying the current screen contents to a private buffer and blending between that and the new contents.

GEM based drivers should call `drm_gem_handle_create()` to create the handle.

RETURNS:

0 on success or a negative error code on failure.

dirty

Optional callback for the dirty fb IOCTL.

Userspace can notify the driver via this callback that an area of the framebuffer has changed and should be flushed to the display hardware. This can also be used internally, e.g. by the fbdev emulation, though that's not the case currently.

See documentation in `drm_mode.h` for the struct `drm_mode_fb_dirty_cmd` for more information as all the semantics and arguments have a one to one mapping on this function.

Atomic drivers should use `drm_atomic_helper_dirtyfb()` to implement this hook.

RETURNS:

0 on success or a negative error code on failure.

struct **drm_framebuffer**

frame buffer object

Definition:

```
struct drm_framebuffer {
    struct drm_device *dev;
    struct list_head head;
    struct drm_mode_object base;
    char comm[TASK_COMM_LEN];
    const struct drm_format_info *format;
    const struct drm_framebuffer_funcs *funcs;
    unsigned int pitches[DRM_FORMAT_MAX_PLANES];
    unsigned int offsets[DRM_FORMAT_MAX_PLANES];
    uint64_t modifier;
    unsigned int width;
    unsigned int height;
    int flags;
    int hot_x;
    int hot_y;
    struct list_head filp_head;
    struct drm_gem_object *obj[DRM_FORMAT_MAX_PLANES];
};
```

Members

dev

DRM device this framebuffer belongs to

head

Place on the *drm_mode_config.fb_list*, access protected by *drm_mode_config.fb_lock*.

base

base modeset object structure, contains the reference count.

comm

Name of the process allocating the fb, used for fb dumping.

format

framebuffer format information

funcs

framebuffer vfunc table

pitches

Line stride per buffer. For userspace created object this is copied from *drm_mode_fb_cmd2*.

offsets

Offset from buffer start to the actual pixel data in bytes, per buffer. For userspace created object this is copied from *drm_mode_fb_cmd2*.

Note that this is a linear offset and does not take into account tiling or buffer layout per **modifier**. It is meant to be used when the actual pixel data for this framebuffer plane starts at an offset, e.g. when multiple planes are allocated within the same backing storage buffer

object. For tiled layouts this generally means its **offsets** must at least be tile-size aligned, but hardware often has stricter requirements.

This should not be used to specify x/y pixel offsets into the buffer data (even for linear buffers). Specifying an x/y pixel offset is instead done through the source rectangle in *struct drm_plane_state*.

modifier

Data layout modifier. This is used to describe tiling, or also special layouts (like compression) of auxiliary buffers. For userspace created object this is copied from *drm_mode_fb_cmd2*.

width

Logical width of the visible area of the framebuffer, in pixels.

height

Logical height of the visible area of the framebuffer, in pixels.

flags

Framebuffer flags like *DRM_MODE_FB_INTERLACED* or *DRM_MODE_FB_MODIFIERS*.

hot_x

X coordinate of the cursor hotspot. Used by the legacy cursor IOCTL when the driver supports cursor through a *DRM_PLANE_TYPE_CURSOR* universal plane.

hot_y

Y coordinate of the cursor hotspot. Used by the legacy cursor IOCTL when the driver supports cursor through a *DRM_PLANE_TYPE_CURSOR* universal plane.

filp_head

Placed on *drm_file.fbs*, protected by *drm_file.fbs_lock*.

obj

GEM objects backing the framebuffer, one per plane (optional).

This is used by the GEM framebuffer helpers, see e.g. *drm_gem_fb_create()*.

Description

Note that the fb is refcounted for the benefit of driver internals, for example some hw, disabling a CRTC/plane is asynchronous, and scanout does not actually complete until the next vblank. So some cleanup (like releasing the reference(s) on the backing GEM bo(s)) should be deferred. In cases like this, the driver would like to hold a ref to the fb even though it has already been removed from userspace perspective. See *drm_framebuffer_get()* and *drm_framebuffer_put()*.

The refcount is stored inside the mode object **base**.

```
void drm_framebuffer_get(struct drm_framebuffer *fb)
    acquire a framebuffer reference
```

Parameters

```
struct drm_framebuffer *fb
    DRM framebuffer
```

Description

This function increments the framebuffer's reference count.

void **drm_framebuffer_put**(struct *drm_framebuffer* *fb)
release a framebuffer reference

Parameters

struct drm_framebuffer *fb
DRM framebuffer

Description

This function decrements the framebuffer's reference count and frees the framebuffer if the reference count drops to zero.

uint32_t **drm_framebuffer_read_refcount**(const struct *drm_framebuffer* *fb)
read the framebuffer reference count.

Parameters

const struct drm_framebuffer *fb
framebuffer

Description

This functions returns the framebuffer's reference count.

void **drm_framebuffer_assign**(struct *drm_framebuffer* **p, struct *drm_framebuffer* *fb)
store a reference to the fb

Parameters

struct drm_framebuffer **p
location to store framebuffer

struct drm_framebuffer *fb
new framebuffer (maybe NULL)

Description

This functions sets the location to store a reference to the framebuffer, unreferencing the framebuffer that was previously stored in that location.

struct **drm_afbc_framebuffer**
a special afbc frame buffer object

Definition:

```
struct drm_afbc_framebuffer {
    struct drm_framebuffer base;
    u32 block_width;
    u32 block_height;
    u32 aligned_width;
    u32 aligned_height;
    u32 offset;
    u32 afbc_size;
};
```

Members

base
base framebuffer structure.

block_width

width of a single afbc block

block_height

height of a single afbc block

aligned_width

aligned frame buffer width

aligned_height

aligned frame buffer height

offset

offset of the first afbc header

afbc_size

minimum size of afbc buffer

Description

A derived class of *struct* `drm_framebuffer`, dedicated for afbc use cases.

```
int drm_framebuffer_init(struct drm_device *dev, struct drm_framebuffer *fb, const struct  
                        drm_framebuffer_funcs *funcs)
```

initialize a framebuffer

Parameters

```
struct drm_device *dev
```

DRM device

```
struct drm_framebuffer *fb
```

framebuffer to be initialized

```
const struct drm_framebuffer_funcs *funcs
```

... with these functions

Description

Allocates an ID for the framebuffer's parent mode object, sets its mode functions & device file and adds it to the master fd list.

IMPORTANT: This functions publishes the fb and makes it available for concurrent access by other users. Which means by this point the fb `_must_` be fully set up - since all the fb attributes are invariant over its lifetime, no further locking but only correct reference counting is required.

Return

Zero on success, error code on failure.

```
struct drm_framebuffer *drm_framebuffer_lookup(struct drm_device *dev, struct drm_file  
                                              *file_priv, uint32_t id)
```

look up a drm framebuffer and grab a reference

Parameters

```
struct drm_device *dev
```

drm device

```
struct drm_file *file_priv
```

drm file to check for lease against.

uint32_t id

id of the fb object

Description

If successful, this grabs an additional reference to the framebuffer - callers need to make sure to eventually unreference the returned framebuffer again, using [drm_framebuffer_put\(\)](#).

void **drm_framebuffer_unregister_private**(struct [drm_framebuffer](#) *fb)

unregister a private fb from the lookup idr

Parameters

struct drm_framebuffer *fb

fb to unregister

Description

Drivers need to call this when cleaning up driver-private framebuffers, e.g. those used for fbdev. Note that the caller must hold a reference of its own, i.e. the object may not be destroyed through this call (since it'll lead to a locking inversion).

NOTE

This function is deprecated. For driver-private framebuffers it is not recommended to embed a framebuffer struct into fbdev struct, instead, a framebuffer pointer is preferred and [drm_framebuffer_put\(\)](#) should be called when the framebuffer is to be cleaned up.

void **drm_framebuffer_cleanup**(struct [drm_framebuffer](#) *fb)

remove a framebuffer object

Parameters

struct drm_framebuffer *fb

framebuffer to remove

Description

Cleanup framebuffer. This function is intended to be used from the drivers [drm_framebuffer_funcs.destroy](#) callback. It can also be used to clean up driver private framebuffers embedded into a larger structure.

Note that this function does not remove the fb from active usage - if it is still used anywhere, hilarity can ensue since userspace could call getfb on the id and get back -EINVAL. Obviously no concern at driver unload time.

Also, the framebuffer will not be removed from the lookup idr - for user-created framebuffers this will happen in the rmfb ioctl. For driver-private objects (e.g. for fbdev) drivers need to explicitly call [drm_framebuffer_unregister_private](#).

void **drm_framebuffer_remove**(struct [drm_framebuffer](#) *fb)

remove and unreference a framebuffer object

Parameters

struct drm_framebuffer *fb

framebuffer to remove

Description

Scans all the CRTC's and planes in **dev**'s mode_config. If they're using **fb**, removes it, setting it to NULL. Then drops the reference to the passed-in framebuffer. Might take the modeset locks.

Note that this function optimizes the cleanup away if the caller holds the last reference to the framebuffer. It is also guaranteed to not take the modeset locks in this case.

int **drm_framebuffer_plane_width**(int width, const struct *drm_framebuffer* *fb, int plane)
width of the plane given the first plane

Parameters

int **width**
width of the first plane

const struct **drm_framebuffer** *fb
the framebuffer

int **plane**
plane index

Return

The width of **plane**, given that the width of the first plane is **width**.

int **drm_framebuffer_plane_height**(int height, const struct *drm_framebuffer* *fb, int plane)
height of the plane given the first plane

Parameters

int **height**
height of the first plane

const struct **drm_framebuffer** *fb
the framebuffer

int **plane**
plane index

Return

The height of **plane**, given that the height of the first plane is **height**.

4.7 DRM Format Handling

In the DRM subsystem, framebuffer pixel formats are described using the fourcc codes defined in *include/uapi/drm/drm_fourcc.h*. In addition to the fourcc code, a Format Modifier may optionally be provided, in order to further describe the buffer's format - for example tiling or compression.

4.7.1 Format Modifiers

Format modifiers are used in conjunction with a fourcc code, forming a unique fourcc:modifier pair. This format:modifier pair must fully define the format and data layout of the buffer, and should be the only way to describe that particular buffer.

Having multiple fourcc:modifier pairs which describe the same layout should be avoided, as such aliases run the risk of different drivers exposing different names for the same data format, forcing userspace to understand that they are aliases.

Format modifiers may change any property of the buffer, including the number of planes and/or the required allocation size. Format modifiers are vendor-namespaced, and as such the relationship between a fourcc code and a modifier is specific to the modifier being used. For example, some modifiers may preserve meaning - such as number of planes - from the fourcc code, whereas others may not.

Modifiers must uniquely encode buffer layout. In other words, a buffer must match only a single modifier. A modifier must not be a subset of layouts of another modifier. For instance, it's incorrect to encode pitch alignment in a modifier: a buffer may match a 64-pixel aligned modifier and a 32-pixel aligned modifier. That said, modifiers can have implicit minimal requirements.

For modifiers where the combination of fourcc code and modifier can alias, a canonical pair needs to be defined and used by all drivers. Preferred combinations are also encouraged where all combinations might lead to confusion and unnecessarily reduced interoperability. An example for the latter is AFBC, where the ABGR layouts are preferred over ARGB layouts.

There are two kinds of modifier users:

- Kernel and user-space drivers: for drivers it's important that modifiers don't alias, otherwise two drivers might support the same format but use different aliases, preventing them from sharing buffers in an efficient format.
- Higher-level programs interfacing with KMS/GBM/EGL/Vulkan/etc: these users see modifiers as opaque tokens they can check for equality and intersect. These users mustn't need to know to reason about the modifier value (i.e. they are not expected to extract information out of the modifier).

Vendors should document their modifier usage in as much detail as possible, to ensure maximum compatibility across devices, drivers and applications.

The authoritative list of format modifier codes is found in *include/uapi/drm/drm_fourcc.h*

4.7.2 Open Source User Waiver

Because this is the authoritative source for pixel formats and modifiers referenced by GL, Vulkan extensions and other standards and hence used both by open source and closed source driver stacks, the usual requirement for an upstream in-kernel or open source userspace user does not apply.

To ensure, as much as feasible, compatibility across stacks and avoid confusion with incompatible enumerations stakeholders for all relevant driver stacks should approve additions.

4.7.3 Format Functions Reference

DRM_FORMAT_MAX_PLANES

DRM_FORMAT_MAX_PLANES ()

maximum number of planes a DRM format can have

Parameters

struct **drm_format_info**

information about a DRM format

Definition:

```
struct drm_format_info {
    u32 format;
    u8 depth;
    u8 num_planes;
    union {
        u8 cpp[DRM_FORMAT_MAX_PLANES];
        u8 char_per_block[DRM_FORMAT_MAX_PLANES];
    };
    u8 block_w[DRM_FORMAT_MAX_PLANES];
    u8 block_h[DRM_FORMAT_MAX_PLANES];
    u8 hsub;
    u8 vsub;
    bool has_alpha;
    bool is_yuv;
    bool is_color_indexed;
};
```

Members

format

4CC format identifier (DRM_FORMAT_*)

depth

Color depth (number of bits per pixel excluding padding bits), valid for a subset of RGB formats only. This is a legacy field, do not use in new code and set to 0 for new formats.

num_planes

Number of color planes (1 to 3)

{unnamed_union}

anonymous

cpp

Number of bytes per pixel (per plane), this is aliased with **char_per_block**. It is deprecated in favour of using the triplet **char_per_block**, **block_w**, **block_h** for better describing the pixel format.

char_per_block

Number of bytes per block (per plane), where blocks are defined as a rectangle of pixels which are stored next to each other in a byte aligned memory region. Together with **block_w** and **block_h** this is used to properly describe tiles in tiled formats or to describe

groups of pixels in packed formats for which the memory needed for a single pixel is not byte aligned.

cpp has been kept for historical reasons because there are a lot of places in drivers where it's used. In drm core for generic code paths the preferred way is to use **char_per_block**, *drm_format_info_block_width()* and *drm_format_info_block_height()* which allows handling both block and non-block formats in the same way.

For formats that are intended to be used only with non-linear modifiers both **cpp** and **char_per_block** must be 0 in the generic format table. Drivers could supply accurate information from their *drm_mode_config.get_format_info* hook if they want the core to be validating the pitch.

block_w

Block width in pixels, this is intended to be accessed through *drm_format_info_block_width()*

block_h

Block height in pixels, this is intended to be accessed through *drm_format_info_block_height()*

hsub

Horizontal chroma subsampling factor

vsub

Vertical chroma subsampling factor

has_alpha

Does the format embeds an alpha component?

is_yuv

Is it a YUV format?

is_color_indexed

Is it a color-indexed format?

bool **drm_format_info_is_yuv_packed**(const struct *drm_format_info* *info)

check that the format info matches a YUV format with data laid in a single plane

Parameters

const struct *drm_format_info* *info

format info

Return

A boolean indicating whether the format info matches a packed YUV format.

bool **drm_format_info_is_yuv_semiplanar**(const struct *drm_format_info* *info)

check that the format info matches a YUV format with data laid in two planes (luminance and chrominance)

Parameters

const struct *drm_format_info* *info

format info

Return

A boolean indicating whether the format info matches a semiplanar YUV format.

bool **drm_format_info_is_yuv_planar**(const struct *drm_format_info* *info)
check that the format info matches a YUV format with data laid in three planes (one for each YUV component)

Parameters

const struct **drm_format_info** *info
format info

Return

A boolean indicating whether the format info matches a planar YUV format.

bool **drm_format_info_is_yuv_sampling_410**(const struct *drm_format_info* *info)
check that the format info matches a YUV format with 4:1:0 sub-sampling

Parameters

const struct **drm_format_info** *info
format info

Return

A boolean indicating whether the format info matches a YUV format with 4:1:0 sub-sampling.

bool **drm_format_info_is_yuv_sampling_411**(const struct *drm_format_info* *info)
check that the format info matches a YUV format with 4:1:1 sub-sampling

Parameters

const struct **drm_format_info** *info
format info

Return

A boolean indicating whether the format info matches a YUV format with 4:1:1 sub-sampling.

bool **drm_format_info_is_yuv_sampling_420**(const struct *drm_format_info* *info)
check that the format info matches a YUV format with 4:2:0 sub-sampling

Parameters

const struct **drm_format_info** *info
format info

Return

A boolean indicating whether the format info matches a YUV format with 4:2:0 sub-sampling.

bool **drm_format_info_is_yuv_sampling_422**(const struct *drm_format_info* *info)
check that the format info matches a YUV format with 4:2:2 sub-sampling

Parameters

const struct **drm_format_info** *info
format info

Return

A boolean indicating whether the format info matches a YUV format with 4:2:2 sub-sampling.

bool **drm_format_info_is_yuv_sampling_444**(const struct *drm_format_info* *info)
check that the format info matches a YUV format with 4:4:4 sub-sampling

Parameters

const struct *drm_format_info* *info
format info

Return

A boolean indicating whether the format info matches a YUV format with 4:4:4 sub-sampling.

int **drm_format_info_plane_width**(const struct *drm_format_info* *info, int width, int plane)
width of the plane given the first plane

Parameters

const struct *drm_format_info* *info
pixel format info

int width
width of the first plane

int plane
plane index

Return

The width of **plane**, given that the width of the first plane is **width**.

int **drm_format_info_plane_height**(const struct *drm_format_info* *info, int height, int plane)
height of the plane given the first plane

Parameters

const struct *drm_format_info* *info
pixel format info

int height
height of the first plane

int plane
plane index

Return

The height of **plane**, given that the height of the first plane is **height**.

uint32_t **drm_mode_legacy_fb_format**(uint32_t bpp, uint32_t depth)
compute drm fourcc code from legacy description

Parameters

uint32_t bpp
bits per pixels

uint32_t depth
bit depth per pixel

Description

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

`uint32_t drm_driver_legacy_fb_format(struct drm_device *dev, uint32_t bpp, uint32_t depth)`

compute drm fourcc code from legacy description

Parameters

`struct drm_device *dev`
DRM device

`uint32_t bpp`
bits per pixels

`uint32_t depth`
bit depth per pixel

Description

Computes a drm fourcc pixel format code for the given **bpp/depth** values. Unlike *drm_mode_legacy_fb_format()* this looks at the drivers mode_config, and depending on the *drm_mode_config.quirk_addfb_prefer_host_byte_order* flag it returns little endian byte order or host byte order framebuffer formats.

`const struct drm_format_info *drm_format_info(u32 format)`
query information for a given format

Parameters

`u32 format`
pixel format (DRM_FORMAT_*)

Description

The caller should only pass a supported pixel format to this function. Unsupported pixel formats will generate a warning in the kernel log.

Return

The instance of *struct drm_format_info* that describes the pixel format, or NULL if the format is unsupported.

`const struct drm_format_info *drm_get_format_info(struct drm_device *dev, const struct drm_mode_fb_cmd2 *mode_cmd)`
query information for a given framebuffer configuration

Parameters

`struct drm_device *dev`
DRM device

`const struct drm_mode_fb_cmd2 *mode_cmd`
metadata from the userspace fb creation request

Return

The instance of *struct drm_format_info* that describes the pixel format, or NULL if the format is unsupported.

`unsigned int drm_format_info_block_width(const struct drm_format_info *info, int plane)`
width in pixels of block.

Parameters

const struct drm_format_info *info

pixel format info

int plane

plane index

Return

The width in pixels of a block, depending on the plane index.

unsigned int **drm_format_info_block_height**(const struct *drm_format_info* *info, int plane)

height in pixels of a block

Parameters

const struct drm_format_info *info

pixel format info

int plane

plane index

Return

The height in pixels of a block, depending on the plane index.

unsigned int **drm_format_info_bpp**(const struct *drm_format_info* *info, int plane)

number of bits per pixel

Parameters

const struct drm_format_info *info

pixel format info

int plane

plane index

Return

The actual number of bits per pixel, depending on the plane index.

uint64_t **drm_format_info_min_pitch**(const struct *drm_format_info* *info, int plane, unsigned int buffer_width)

computes the minimum required pitch in bytes

Parameters

const struct drm_format_info *info

pixel format info

int plane

plane index

unsigned int buffer_width

buffer width in pixels

Return

The minimum required pitch in bytes for a buffer by taking into consideration the pixel format information and the buffer width.

4.8 Dumb Buffer Objects

The KMS API doesn't standardize backing storage object creation and leaves it to driver-specific ioctls. Furthermore actually creating a buffer object even for GEM-based drivers is done through a driver-specific ioctl - GEM only has a common userspace interface for sharing and destroying objects. While not an issue for full-fledged graphics stacks that include device-specific userspace components (in libdrm for instance), this limit makes DRM-based early boot graphics unnecessarily complex.

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

To support dumb objects drivers must implement the `drm_driver.dumb_create` and `drm_driver.dumb_map_offset` operations (the latter defaults to `drm_gem_dumb_map_offset()` if not set). Drivers that don't use GEM handles additionally need to implement the `drm_driver.dumb_destroy` operation. See the callbacks for further details.

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

4.9 Plane Abstraction

A plane represents an image source that can be blended with or overlaid on top of a CRTC during the scanout process. Planes take their input data from a `drm_framebuffer` object. The plane itself specifies the cropping and scaling of that image, and where it is placed on the visible area of a display pipeline, represented by `drm_crtc`. A plane can also have additional properties that specify how the pixels are positioned and blended, like rotation or Z-position. All these properties are stored in `drm_plane_state`.

Unless explicitly specified (via CRTC property or otherwise), the active area of a CRTC will be black by default. This means portions of the active area which are not covered by a plane will be black, and alpha blending of any planes with the CRTC background will blend with black at the lowest zpos.

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

Each plane has a type, see `enum drm_plane_type`. A plane can be compatible with multiple CRTCs, see `drm_plane.possible_crtcs`.

Each CRTC must have a unique primary plane userspace can attach to enable the CRTC. In other words, userspace must be able to attach a different primary plane to each CRTC at the same time. Primary planes can still be compatible with multiple CRTCs. There must be exactly as many primary planes as there are CRTCs.

Legacy uAPI doesn't expose the primary and cursor planes directly. DRM core relies on the driver to set the primary and optionally the cursor plane used for legacy IOCTLs. This is done by calling `drm_crtc_init_with_planes()`. All drivers must provide one primary plane per CRTC to avoid surprising legacy userspace too much.

4.9.1 Plane Functions Reference

struct **drm_plane_state**
mutable plane state

Definition:

```
struct drm_plane_state {
    struct drm_plane *plane;
    struct drm_crtc *crtc;
    struct drm_framebuffer *fb;
    struct dma_fence *fence;
    int32_t crtc_x;
    int32_t crtc_y;
    uint32_t crtc_w, crtc_h;
    uint32_t src_x;
    uint32_t src_y;
    uint32_t src_h, src_w;
    ul6 alpha;
    uint16_t pixel_blend_mode;
    unsigned int rotation;
    unsigned int zpos;
    unsigned int normalized_zpos;
    enum drm_color_encoding color_encoding;
    enum drm_color_range color_range;
    struct drm_property_blob *fb_damage_clips;
    bool ignore_damage_clips;
    struct drm_rect src, dst;
    bool visible;
    enum drm_scaling_filter scaling_filter;
    struct drm_crtc_commit *commit;
    struct drm_atomic_state *state;
};
```

Members

plane

backpointer to the plane

crtc

Currently bound CRTC, NULL if disabled. Do not write this directly, use [drm_atomic_set_crtc_for_plane\(\)](#)

fb

Currently bound framebuffer. Do not write this directly, use [drm_atomic_set_fb_for_plane\(\)](#)

fence

Optional fence to wait for before scanning out **fb**. The core atomic code will set this when userspace is using explicit fencing. Do not write this field directly for a driver's implicit fence.

Drivers should store any implicit fence in this from their [drm_plane_helper_funcs.prepare_fb](#) callback. See [drm_gem_plane_helper_prepare_fb\(\)](#) for a suitable helper.

crtc_x

Left position of visible portion of plane on crtc, signed dest location allows it to be partially off screen.

crtc_y

Upper position of visible portion of plane on crtc, signed dest location allows it to be partially off screen.

crtc_w

width of visible portion of plane on crtc

crtc_h

height of visible portion of plane on crtc

src_x

left position of visible portion of plane within plane (in 16.16 fixed point).

src_y

upper position of visible portion of plane within plane (in 16.16 fixed point).

src_h

height of visible portion of plane (in 16.16)

src_w

width of visible portion of plane (in 16.16)

alpha

Opacity of the plane with 0 as completely transparent and 0xffff as completely opaque. See [drm_plane_create_alpha_property\(\)](#) for more details.

pixel_blend_mode

The alpha blending equation selection, describing how the pixels from the current plane are composited with the background. Value can be one of `DRM_MODE_BLEND_*`

rotation

Rotation of the plane. See [drm_plane_create_rotation_property\(\)](#) for more details.

zpos

Priority of the given plane on crtc (optional).

User-space may set mutable zpos properties so that multiple active planes on the same CRTC have identical zpos values. This is a user-space bug, but drivers can solve the conflict by comparing the plane object IDs; the plane with a higher ID is stacked on top of a plane with a lower ID.

See [drm_plane_create_zpos_property\(\)](#) and [drm_plane_create_zpos_immutable_property\(\)](#) for more details.

normalized_zpos

Normalized value of zpos: unique, range from 0 to N-1 where N is the number of active planes for given crtc. Note that the driver must set [drm_mode_config.normalize_zpos](#) or call [drm_atomic_normalize_zpos\(\)](#) to update this before it can be trusted.

color_encoding

Color encoding for non RGB formats

color_range

Color range for non RGB formats

fb_damage_clips

Blob representing damage (area in plane framebuffer that changed since last plane update) as an array of `drm_mode_rect` in framebuffer coordinates of the attached framebuffer. Note that unlike plane src, damage clips are not in 16.16 fixed point.

See `drm_plane_get_damage_clips()` and `drm_plane_get_damage_clips_count()` for accessing these.

ignore_damage_clips

Set by drivers to indicate the `drm_atomic_helper_damage_iter_init()` helper that the **fb_damage_clips** blob property should be ignored.

See *Damage Tracking Properties* for more information.

src

source coordinates of the plane (in 16.16).

When using `drm_atomic_helper_check_plane_state()`, the coordinates are clipped, but the driver may choose to use unclipped coordinates instead when the hardware performs the clipping automatically.

dst

clipped destination coordinates of the plane.

When using `drm_atomic_helper_check_plane_state()`, the coordinates are clipped, but the driver may choose to use unclipped coordinates instead when the hardware performs the clipping automatically.

visible

Visibility of the plane. This can be false even if `fb!=NULL` and `crtc!=NULL`, due to clipping.

scaling_filter

Scaling filter to be applied

commit

Tracks the pending commit to prevent use-after-free conditions, and for async plane updates.

May be NULL.

state

backpointer to global `drm_atomic_state`

Description

Please note that the destination coordinates **crtc_x**, **crtc_y**, **crtc_h** and **crtc_w** and the source coordinates **src_x**, **src_y**, **src_h** and **src_w** are the raw coordinates provided by userspace. Drivers should use `drm_atomic_helper_check_plane_state()` and only use the derived rectangles in **src** and **dst** to program the hardware.

struct drm_plane_funcs

driver plane control functions

Definition:

```
struct drm_plane_funcs {
    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, struct
```

```

↪drm_modeset_acquire_ctx *ctx);
    int (*disable_plane)(struct drm_plane *plane, struct drm_modeset_acquire_
↪ctx *ctx);
    void (*destroy)(struct drm_plane *plane);
    void (*reset)(struct drm_plane *plane);
    int (*set_property)(struct drm_plane *plane, struct drm_property *property,
↪ uint64_t val);
    struct drm_plane_state *(*atomic_duplicate_state)(struct drm_plane *plane);
    void (*atomic_destroy_state)(struct drm_plane *plane, struct drm_plane_
↪state *state);
    int (*atomic_set_property)(struct drm_plane *plane, struct drm_plane_state
↪*state, struct drm_property *property, uint64_t val);
    int (*atomic_get_property)(struct drm_plane *plane, const struct drm_plane_
↪state *state, struct drm_property *property, uint64_t *val);
    int (*late_register)(struct drm_plane *plane);
    void (*early_unregister)(struct drm_plane *plane);
    void (*atomic_print_state)(struct drm_printer *p, const struct drm_plane_
↪state *state);
    bool (*format_mod_supported)(struct drm_plane *plane, uint32_t format,
↪uint64_t modifier);
};

```

Members

update_plane

This is the legacy entry point to enable and configure the plane for the given CRTC and framebuffer. It is never called to disable the plane, i.e. the passed-in crtc and fb parameters are never NULL.

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.

Drivers implementing atomic modeset should use [`drm_atomic_helper_update_plane\(\)`](#) to implement this hook.

RETURNS:

0 on success or a negative error code on failure.

disable_plane

This is the legacy entry point to disable the plane. The DRM core calls this method in response to a `DRM_IOCTL_MODE_SETPANE` IOCTL call with the frame buffer ID set to 0. Disabled planes must not be processed by the CRTC.

Drivers implementing atomic modeset should use [`drm_atomic_helper_disable_plane\(\)`](#) to implement this hook.

RETURNS:

0 on success or a negative error code on failure.

destroy

Clean up plane resources. This is only called at driver unload time through `drm_mode_config_cleanup()` since a plane cannot be hotplugged in DRM.

reset

Reset plane hardware and software state to off. This function isn't called by the core directly, only through `drm_mode_config_reset()`. It's not a helper hook only for historical reasons.

Atomic drivers can use `drm_atomic_helper_plane_reset()` to reset atomic state using this hook.

set_property

This is the legacy entry point to update a property attached to the plane.

This callback is optional if the driver does not support any legacy driver-private properties. For atomic drivers it is not used because property handling is done entirely in the DRM core.

RETURNS:

0 on success or a negative error code on failure.

atomic_duplicate_state

Duplicate the current atomic state for this plane and return it. The core and helpers guarantee that any atomic state duplicated with this hook and still owned by the caller (i.e. not transferred to the driver by calling `drm_mode_config_funcs.atomic_commit`) will be cleaned up by calling the **atomic_destroy_state** hook in this structure.

This callback is mandatory for atomic drivers.

Atomic drivers which don't subclass `struct drm_plane_state` should use `drm_atomic_helper_plane_duplicate_state()`. Drivers that subclass the state structure to extend it with driver-private state should use `__drm_atomic_helper_plane_duplicate_state()` to make sure shared state is duplicated in a consistent fashion across drivers.

It is an error to call this hook before `drm_plane.state` has been initialized correctly.

NOTE:

If the duplicate state references refcounted resources this hook must acquire a reference for each of them. The driver must release these references again in **atomic_destroy_state**.

RETURNS:

Duplicated atomic state or NULL when the allocation failed.

atomic_destroy_state

Destroy a state duplicated with **atomic_duplicate_state** and release or unreference all resources it references

This callback is mandatory for atomic drivers.

atomic_set_property

Decode a driver-private property value and store the decoded value into the passed-in state structure. Since the atomic core decodes all standardized properties (even for extensions

beyond the core set of properties which might not be implemented by all drivers) this requires drivers to subclass the state structure.

Such driver-private properties should really only be implemented for truly hardware/vendor specific state. Instead it is preferred to standardize atomic extension and decode the properties used to expose such an extension in the core.

Do not call this function directly, use `drm_atomic_plane_set_property()` instead.

This callback is optional if the driver does not support any driver-private atomic properties.

NOTE:

This function is called in the state assembly phase of atomic modesets, which can be aborted for any reason (including on userspace's request to just check whether a configuration would be possible). Drivers **MUST NOT** touch any persistent state (hardware or software) or data structures except the passed in **state** parameter.

Also since userspace controls in which order properties are set this function must not do any input validation (since the state update is incomplete and hence likely inconsistent). Instead any such input validation must be done in the various `atomic_check` callbacks.

RETURNS:

0 if the property has been found, `-EINVAL` if the property isn't implemented by the driver (which shouldn't ever happen, the core only asks for properties attached to this plane). No other validation is allowed by the driver. The core already checks that the property value is within the range (integer, valid enum value, ...) the driver set when registering the property.

atomic_get_property

Reads out the decoded driver-private property. This is used to implement the `GETPLANE` IOCTL.

Do not call this function directly, use `drm_atomic_plane_get_property()` instead.

This callback is optional if the driver does not support any driver-private atomic properties.

RETURNS:

0 on success, `-EINVAL` if the property isn't implemented by the driver (which should never happen, the core only asks for properties attached to this plane).

late_register

This optional hook can be used to register additional userspace interfaces attached to the plane like debugfs interfaces. It is called late in the driver load sequence from `drm_dev_register()`. Everything added from this callback should be unregistered in the `early_unregister` callback.

Returns:

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

early_unregister

This optional hook should be used to unregister the additional userspace interfaces attached to the plane from **late_register**. It is called from `drm_dev_unregister()`, early in the driver unload sequence to disable userspace access before data structures are torn-down.

atomic_print_state

If driver subclasses *struct drm_plane_state*, it should implement this optional hook for printing additional driver specific state.

Do not call this directly, use `drm_atomic_plane_print_state()` instead.

format_mod_supported

This optional hook is used for the DRM to determine if the given format/modifier combination is valid for the plane. This allows the DRM to generate the correct format bitmask (which formats apply to which modifier), and to validate modifiers at `atomic_check` time.

If not present, then any modifier in the plane's modifier list is allowed with any of the plane's formats.

Returns:

True if the given modifier is valid for that format on the plane. False otherwise.

enum drm_plane_type

uapi plane type enumeration

Constants

DRM_PLANE_TYPE_OVERLAY

Overlay planes represent all non-primary, non-cursor planes. Some drivers refer to these types of planes as “sprites” internally.

DRM_PLANE_TYPE_PRIMARY

A primary plane attached to a CRTC is the most likely to be able to light up the CRTC when no scaling/cropping is used and the plane covers the whole CRTC.

DRM_PLANE_TYPE_CURSOR

A cursor plane attached to a CRTC is more likely to be able to be enabled when no scaling/cropping is used and the framebuffer has the size indicated by *drm_mode_config.cursor_width* and *drm_mode_config.cursor_height*. Additionally, if the driver doesn't support modifiers, the framebuffer should have a linear layout.

Description

For historical reasons not all planes are made the same. This enumeration is used to tell the different types of planes apart to implement the different uapi semantics for them. For userspace which is universal plane aware and which is using that atomic IOCTL there's no difference between these planes (beyond what the driver and hardware can support of course).

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

In addition to setting each plane's type, drivers need to setup the *drm_crtc.primary* and optionally *drm_crtc.cursor* pointers for legacy IOCTLs. See *drm_crtc_init_with_planes()*.

WARNING: The values of this enum is UABI since they're exposed in the “type” property.

struct drm_plane

central DRM plane control structure

Definition:

```

struct drm_plane {
    struct drm_device *dev;
    struct list_head head;
    char *name;
    struct drm_modeset_lock mutex;
    struct drm_mode_object base;
    uint32_t possible_crtcs;
    uint32_t *format_types;
    unsigned int format_count;
    bool format_default;
    uint64_t *modifiers;
    unsigned int modifier_count;
    struct drm_crtc *crtc;
    struct drm_framebuffer *fb;
    struct drm_framebuffer *old_fb;
    const struct drm_plane_funcs *funcs;
    struct drm_object_properties properties;
    enum drm_plane_type type;
    unsigned index;
    const struct drm_plane_helper_funcs *helper_private;
    struct drm_plane_state *state;
    struct drm_property *alpha_property;
    struct drm_property *zpos_property;
    struct drm_property *rotation_property;
    struct drm_property *blend_mode_property;
    struct drm_property *color_encoding_property;
    struct drm_property *color_range_property;
    struct drm_property *scaling_filter_property;
};

```

Members

dev

DRM device this plane belongs to

head

List of all planes on **dev**, linked from [drm_mode_config.plane_list](#). Invariant over the lifetime of **dev** and therefore does not need locking.

name

human readable name, can be overwritten by the driver

mutex

Protects modeset plane state, together with the [drm_crtc.mutex](#) of CRTC this plane is linked to (when active, getting activated or getting disabled).

For atomic drivers specifically this protects **state**.

base

base mode object

possible_crtcs

pipes this plane can be bound to constructed from [drm_crtc_mask\(\)](#)

format_types

array of formats supported by this plane

format_count

Size of the array pointed at by **format_types**.

format_default

driver hasn't supplied supported formats for the plane. Used by the non-atomic driver compatibility wrapper only.

modifiers

array of modifiers supported by this plane

modifier_count

Size of the array pointed at by **modifier_count**.

crtc

Currently bound CRTC, only meaningful for non-atomic drivers. For atomic drivers this is forced to be NULL, atomic drivers should instead check [*drm_plane_state.crtc*](#).

fb

Currently bound framebuffer, only meaningful for non-atomic drivers. For atomic drivers this is forced to be NULL, atomic drivers should instead check [*drm_plane_state.fb*](#).

old_fb

Temporary tracking of the old fb while a modeset is ongoing. Only used by non-atomic drivers, forced to be NULL for atomic drivers.

funcs

plane control functions

properties

property tracking for this plane

type

Type of plane, see [*enum drm_plane_type*](#) for details.

index

Position inside the `mode_config.list`, can be used as an array index. It is invariant over the lifetime of the plane.

helper_private

mid-layer private data

state

Current atomic state for this plane.

This is protected by **mutex**. Note that nonblocking atomic commits access the current plane state without taking locks. Either by going through the [*struct drm_atomic_state*](#) pointers, see [*for_each_oldnew_plane_in_state\(\)*](#), [*for_each_old_plane_in_state\(\)*](#) and [*for_each_new_plane_in_state\(\)*](#). Or through careful ordering of atomic commit operations as implemented in the atomic helpers, see [*struct drm_crtc_commit*](#).

alpha_property

Optional alpha property for this plane. See [*drm_plane_create_alpha_property\(\)*](#).

zpos_property

Optional zpos property for this plane. See [*drm_plane_create_zpos_property\(\)*](#).

rotation_property

Optional rotation property for this plane. See [drm_plane_create_rotation_property\(\)](#).

blend_mode_property

Optional “pixel blend mode” enum property for this plane. Blend mode property represents the alpha blending equation selection, describing how the pixels from the current plane are composited with the background.

color_encoding_property

Optional “COLOR_ENCODING” enum property for specifying color encoding for non RGB formats. See [drm_plane_create_color_properties\(\)](#).

color_range_property

Optional “COLOR_RANGE” enum property for specifying color range for non RGB formats. See [drm_plane_create_color_properties\(\)](#).

scaling_filter_property

property to apply a particular filter while scaling.

Description

Planes represent the scanout hardware of a display block. They receive their input data from a [drm_framebuffer](#) and feed it to a [drm_crtc](#). Planes control the color conversion, see [Plane Composition Properties](#) for more details, and are also involved in the color conversion of input pixels, see [Color Management Properties](#) for details on that.

drm_universal_plane_alloc

`drm_universal_plane_alloc (dev, type, member, possible_crtcs, funcs, formats, format_count, format_modifiers, plane_type, name, ...)`

Allocate and initialize an universal plane object

Parameters**dev**

DRM device

type

the type of the struct which contains struct [drm_plane](#)

member

the name of the [drm_plane](#) within **type**

possible_crtcs

bitmask of possible CRTCs

funcs

callbacks for the new plane

formats

array of supported formats (DRM_FORMAT_*)

format_count

number of elements in **formats**

format_modifiers

array of struct `drm_format` modifiers terminated by `DRM_FORMAT_MOD_INVALID`

plane_type

type of plane (overlay, primary, cursor)

name

printf style format string for the plane name, or NULL for default name

...

variable arguments

Description

Allocates and initializes a plane object of type **type**. Cleanup is automatically handled through registering *drm_plane_cleanup()* with *drmm_add_action()*.

The **drm_plane_funcs.destroy** hook must be NULL.

Drivers that only support the DRM_FORMAT_MOD_LINEAR modifier support may set **format_modifiers** to NULL. The plane will advertise the linear modifier.

Return

Pointer to new plane, or ERR_PTR on failure.

drm_universal_plane_alloc

drm_universal_plane_alloc (dev, type, member, possible_crtcs, funcs, formats, format_count, format_modifiers, plane_type, name, ...)

Allocate and initialize an universal plane object

Parameters**dev**

DRM device

type

the type of the struct which contains struct *drm_plane*

member

the name of the *drm_plane* within **type**

possible_crtcs

bitmask of possible CRTC's

funcs

callbacks for the new plane

formats

array of supported formats (DRM_FORMAT_*)

format_count

number of elements in **formats**

format_modifiers

array of struct drm_format modifiers terminated by DRM_FORMAT_MOD_INVALID

plane_type

type of plane (overlay, primary, cursor)

name

printf style format string for the plane name, or NULL for default name

...

variable arguments

Description

Allocates and initializes a plane object of type **type**. The caller is responsible for releasing the allocated memory with `kfree()`.

Drivers are encouraged to use `drm_universal_plane_alloc()` instead.

Drivers that only support the `DRM_FORMAT_MOD_LINEAR` modifier support may set **format_modifiers** to `NULL`. The plane will advertise the linear modifier.

Return

Pointer to new plane, or `ERR_PTR` on failure.

unsigned int **drm_plane_index**(const struct *drm_plane* *plane)
find the index of a registered plane

Parameters

const struct drm_plane *plane
plane to find index for

Description

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

u32 **drm_plane_mask**(const struct *drm_plane* *plane)
find the mask of a registered plane

Parameters

const struct drm_plane *plane
plane to find mask for

struct *drm_plane* ***drm_plane_find**(struct *drm_device* *dev, struct *drm_file* *file_priv, uint32_t id)
find a *drm_plane*

Parameters

struct drm_device *dev
DRM device

struct drm_file *file_priv
drm file to check for lease against.

uint32_t id
plane id

Description

Returns the plane with **id**, `NULL` if it doesn't exist. Simple wrapper around `drm_mode_object_find()`.

drm_for_each_plane_mask

`drm_for_each_plane_mask` (plane, dev, plane_mask)
iterate over planes specified by bitmask

Parameters

plane

the loop cursor

dev

the DRM device

plane_mask

bitmask of plane indices

Description

Iterate over all planes specified by bitmask.

drm_for_each_legacy_plane

`drm_for_each_legacy_plane (plane, dev)`

iterate over all planes for legacy userspace

Parameters

plane

the loop cursor

dev

the DRM device

Description

Iterate over all legacy planes of **dev**, excluding primary and cursor planes. This is useful for implementing userspace apis when userspace is not universal plane aware. See also [*enum drm_plane_type*](#).

drm_for_each_plane

`drm_for_each_plane (plane, dev)`

iterate over all planes

Parameters

plane

the loop cursor

dev

the DRM device

Description

Iterate over all planes of **dev**, include primary and cursor planes.

```
int drm_universal_plane_init(struct drm\_device *dev, struct drm\_plane *plane, uint32_t
                             possible_crtcs, const struct drm\_plane\_funcs *funcs, const
                             uint32_t *formats, unsigned int format_count, const uint64_t
                             *format_modifiers, enum drm\_plane\_type type, const char
                             *name, ...)
```

Initialize a new universal plane object

Parameters

struct drm_device *dev

DRM device

struct drm_plane *plane
plane object to init

uint32_t possible_crtcs
bitmask of possible CRTCs

const struct drm_plane_funcs *funcs
callbacks for the new plane

const uint32_t *formats
array of supported formats (DRM_FORMAT_*)

unsigned int format_count
number of elements in **formats**

const uint64_t *format_modifiers
array of struct `drm_format` modifiers terminated by `DRM_FORMAT_MOD_INVALID`

enum drm_plane_type type
type of plane (overlay, primary, cursor)

const char *name
printf style format string for the plane name, or NULL for default name

...

variable arguments

Description

Initializes a plane object of type **type**. The `drm_plane_funcs.destroy` hook should call `drm_plane_cleanup()` and `kfree()` the plane structure. The plane structure should not be allocated with `devm_kzalloc()`.

Drivers that only support the `DRM_FORMAT_MOD_LINEAR` modifier support may set **format_modifiers** to NULL. The plane will advertise the linear modifier.

Note

consider using `drm_universal_plane_alloc()` instead of `drm_universal_plane_init()` to let the DRM managed resource infrastructure take care of cleanup and deallocation.

Return

Zero on success, error code on failure.

void **drm_plane_cleanup**(struct *drm_plane* *plane)
Clean up the core plane usage

Parameters

struct drm_plane *plane
plane to cleanup

Description

This function cleans up **plane** and removes it from the DRM mode setting core. Note that the function does *not* free the plane structure itself, this is the responsibility of the caller.

struct *drm_plane* ***drm_plane_from_index**(struct *drm_device* *dev, int idx)
find the registered plane at an index

Parameters

struct drm_device *dev

DRM device

int idx

index of registered plane to find for

Description

Given a plane index, return the registered plane from DRM device's list of planes with matching index. This is the inverse of [drm_plane_index\(\)](#).

void **drm_plane_force_disable**(struct [drm_plane](#) *plane)

Forcibly disable a plane

Parameters

struct drm_plane *plane

plane to disable

Description

Forces the plane to be disabled.

Used when the plane's current framebuffer is destroyed, and when restoring fbdev mode.

Note that this function is not suitable for atomic drivers, since it doesn't wire through the lock acquisition context properly and hence can't handle retries or driver private locks. You probably want to use [drm_atomic_helper_disable_plane\(\)](#) or [drm_atomic_helper_disable_planes_on_crtc\(\)](#) instead.

int **drm_mode_plane_set_obj_prop**(struct [drm_plane](#) *plane, struct [drm_property](#) *property, uint64_t value)

set the value of a property

Parameters

struct drm_plane *plane

drm plane object to set property value for

struct drm_property *property

property to set

uint64_t value

value the property should be set to

Description

This functions sets a given property on a given plane object. This function calls the driver's `->set_property` callback and changes the software state of the property if the callback succeeds.

Return

Zero on success, error code on failure.

bool **drm_any_plane_has_format**(struct [drm_device](#) *dev, u32 format, u64 modifier)

Check whether any plane supports this format and modifier combination

Parameters

struct drm_device *dev

DRM device

u32 format

pixel format (DRM_FORMAT_*)

u64 modifier

data layout modifier

Return

Whether at least one plane supports the specified format and modifier combination.

void **drm_plane_enable_fb_damage_clips**(struct *drm_plane* *plane)

Enables plane fb damage clips property.

Parameters

struct *drm_plane* *plane

Plane on which to enable damage clips property.

Description

This function lets driver to enable the damage clips property on a plane.

unsigned int **drm_plane_get_damage_clips_count**(const struct *drm_plane_state* *state)

Returns damage clips count.

Parameters

const struct *drm_plane_state* *state

Plane state.

Description

Simple helper to get the number of *drm_mode_rect* clips set by user-space during plane update.

Return

Number of clips in plane fb_damage_clips blob property.

struct *drm_mode_rect* ***drm_plane_get_damage_clips**(const struct *drm_plane_state* *state)

Returns damage clips.

Parameters

const struct *drm_plane_state* *state

Plane state.

Description

Note that this function returns uapi type *drm_mode_rect*. Drivers might want to use the helper functions *drm_atomic_helper_damage_iter_init()* and *drm_atomic_helper_damage_iter_next()* or *drm_atomic_helper_damage_merged()* if the driver can only handle a single damage region at most.

Return

Damage clips in plane fb_damage_clips blob property.

int **drm_plane_create_scaling_filter_property**(struct *drm_plane* *plane, unsigned int supported_filters)

create a new scaling filter property

Parameters

struct drm_plane *plane

drm plane

unsigned int supported_filters

bitmask of supported scaling filters, must include BIT(DRM_SCALING_FILTER_DEFAULT).

Description

This function lets driver to enable the scaling filter property on a given plane.

Return

Zero for success or -errno

4.9.2 Plane Composition Functions Reference

int drm_plane_create_alpha_property(struct *drm_plane* *plane)

create a new alpha property

Parameters

struct drm_plane *plane

drm plane

Description

This function creates a generic, mutable, alpha property and enables support for it in the DRM core. It is attached to **plane**.

The alpha property will be allowed to be within the bounds of 0 (transparent) to 0xffff (opaque).

Return

0 on success, negative error code on failure.

int drm_plane_create_rotation_property(struct *drm_plane* *plane, unsigned int rotation, unsigned int supported_rotations)

create a new rotation property

Parameters

struct drm_plane *plane

drm plane

unsigned int rotation

initial value of the rotation property

unsigned int supported_rotations

bitmask of supported rotations and reflections

Description

This creates a new property with the selected support for transformations.

Since a rotation by 180° degress is the same as reflecting both along the x and the y axis the rotation property is somewhat redundant. Drivers can use *drm_rotation_simplify()* to normalize values of this property.

The property exposed to userspace is a bitmask property (see *drm_property_create_bitmask()*) called “rotation” and has the following bitmask enumeration values:

DRM_MODE_ROTATE_0:

"rotate-0"

DRM_MODE_ROTATE_90:

"rotate-90"

DRM_MODE_ROTATE_180:

"rotate-180"

DRM_MODE_ROTATE_270:

"rotate-270"

DRM_MODE_REFLECT_X:

"reflect-x"

DRM_MODE_REFLECT_Y:

"reflect-y"

Rotation is the specified amount in degrees in counter clockwise direction, the X and Y axis are within the source rectangle, i.e. the X/Y axis before rotation. After reflection, the rotation is applied to the image sampled from the source rectangle, before scaling it to fit the destination rectangle.

unsigned int **drm_rotation_simplify**(unsigned int rotation, unsigned int supported_rotations)

Try to simplify the rotation

Parameters

unsigned int rotation

Rotation to be simplified

unsigned int supported_rotations

Supported rotations

Description

Attempt to simplify the rotation to a form that is supported. Eg. if the hardware supports everything except DRM_MODE_REFLECT_X one could call this function like this:

```
drm_rotation_simplify(rotation, DRM_MODE_ROTATE_0 |
    DRM_MODE_ROTATE_90 | DRM_MODE_ROTATE_180 | DRM_MODE_ROTATE_270 |
    DRM_MODE_REFLECT_Y);
```

to eliminate the DRM_MODE_REFLECT_X flag. Depending on what kind of transforms the hardware supports, this function may not be able to produce a supported transform, so the caller should check the result afterwards.

int **drm_plane_create_zpos_property**(struct *drm_plane* *plane, unsigned int zpos, unsigned int min, unsigned int max)

create mutable zpos property

Parameters

struct drm_plane *plane

drm plane

unsigned int zpos

initial value of zpos property

unsigned int min

minimal possible value of zpos property

unsigned int max

maximal possible value of zpos property

Description

This function initializes generic mutable zpos property and enables support for it in drm core. Drivers can then attach this property to planes to enable support for configurable planes arrangement during blending operation. Drivers that attach a mutable zpos property to any plane should call the [drm_atomic_normalize_zpos\(\)](#) helper during their implementation of [drm_mode_config_funcs.atomic_check\(\)](#), which will update the normalized zpos values and store them in [drm_plane_state.normalized_zpos](#). Usually min should be set to 0 and max to maximal number of planes for given crtc - 1.

If zpos of some planes cannot be changed (like fixed background or cursor/topmost planes), drivers shall adjust the min/max values and assign those planes immutable zpos properties with lower or higher values (for more information, see [drm_plane_create_zpos_immutable_property\(\)](#) function). In such case drivers shall also assign proper initial zpos values for all planes in its [plane_reset\(\)](#) callback, so the planes will be always sorted properly.

See also [drm_atomic_normalize_zpos\(\)](#).

The property exposed to userspace is called "zpos".

Return

Zero on success, negative errno on failure.

int **drm_plane_create_zpos_immutable_property**(struct [drm_plane](#) *plane, unsigned int zpos)

create immutable zpos property

Parameters

struct drm_plane *plane

drm plane

unsigned int zpos

value of zpos property

Description

This function initializes generic immutable zpos property and enables support for it in drm core. Using this property driver lets userspace to get the arrangement of the planes for blending operation and notifies it that the hardware (or driver) doesn't support changing of the planes' order. For mutable zpos see [drm_plane_create_zpos_property\(\)](#).

The property exposed to userspace is called "zpos".

Return

Zero on success, negative errno on failure.

int **drm_atomic_normalize_zpos**(struct [drm_device](#) *dev, struct [drm_atomic_state](#) *state)

calculate normalized zpos values for all crtcs

Parameters

struct drm_device *dev
DRM device

struct drm_atomic_state *state
atomic state of DRM device

Description

This function calculates normalized zpos value for all modified planes in the provided atomic state of DRM device.

For every CRTC this function checks new states of all planes assigned to it and calculates normalized zpos value for these planes. Planes are compared first by their zpos values, then by plane id (if zpos is equal). The plane with lowest zpos value is at the bottom. The [drm_plane_state.normalized_zpos](#) is then filled with unique values from 0 to number of active planes in crtc minus one.

RETURNS Zero for success or -errno

int **drm_plane_create_blend_mode_property**(struct [drm_plane](#) *plane, unsigned int supported_modes)

create a new blend mode property

Parameters

struct drm_plane *plane
drm plane

unsigned int supported_modes
bitmask of supported modes, must include BIT(DRM_MODE_BLEND_PREMULTI). Current DRM assumption is that alpha is premultiplied, and old userspace can break if the property defaults to anything else.

Description

This creates a new property describing the blend mode.

The property exposed to userspace is an enumeration property (see [drm_property_create_enum\(\)](#)) called “pixel blend mode” and has the following enumeration values:

“None”:
Blend formula that ignores the pixel alpha.

“Pre-multiplied”:
Blend formula that assumes the pixel color values have been already pre-multiplied with the alpha channel values.

“Coverage”:
Blend formula that assumes the pixel color values have not been pre-multiplied and will do so when blending them to the background color values.

Return

Zero for success or -errno

4.9.3 Plane Damage Tracking Functions Reference

void **drm_atomic_helper_check_plane_damage**(struct *drm_atomic_state* *state, struct *drm_plane_state* *plane_state)

Verify plane damage on atomic_check.

Parameters

struct drm_atomic_state *state

The driver state object.

struct drm_plane_state *plane_state

Plane state for which to verify damage.

Description

This helper function makes sure that damage from plane state is discarded for full modeset. If there are more reasons a driver would want to do a full plane update rather than processing individual damage regions, then those cases should be taken care of here.

Note that *drm_plane_state.fb_damage_clips* == NULL in plane state means that full plane update should happen. It also ensure helper iterator will return *drm_plane_state.src* as damage.

int **drm_atomic_helper_dirtyfb**(struct *drm_framebuffer* *fb, struct *drm_file* *file_priv, unsigned int flags, unsigned int color, struct *drm_clip_rect* *clips, unsigned int num_clips)

Helper for dirtyfb.

Parameters

struct drm_framebuffer *fb

DRM framebuffer.

struct drm_file *file_priv

Drm file for the ioctl call.

unsigned int flags

Dirty fb annotate flags.

unsigned int color

Color for annotate fill.

struct drm_clip_rect *clips

Dirty region.

unsigned int num_clips

Count of clip in clips.

Description

A helper to implement *drm_framebuffer_funcs.dirty* using damage interface during plane update. If num_clips is 0 then this helper will do a full plane update. This is the same behaviour expected by DIRTYFB IOCTL.

Note that this helper is blocking implementation. This is what current drivers and userspace expect in their DIRTYFB IOCTL implementation, as a way to rate-limit userspace and make sure its rendering doesn't get ahead of uploading new data too much.

Return

Zero on success, negative errno on failure.

```
void drm_atomic_helper_damage_iter_init(struct drm_atomic_helper_damage_iter *iter,  
                                         const struct drm_plane_state *old_state, const  
                                         struct drm_plane_state *state)
```

Initialize the damage iterator.

Parameters

struct *drm_atomic_helper_damage_iter* *iter
The iterator to initialize.

const struct *drm_plane_state* *old_state
Old plane state for validation.

const struct *drm_plane_state* *state
Plane state from which to iterate the damage clips.

Description

Initialize an iterator, which clips plane damage *drm_plane_state.fb_damage_clips* to plane *drm_plane_state.src*. This iterator returns full plane src in case damage is not present because either user-space didn't sent or driver discarded it (it want to do full plane update). Currently this iterator returns full plane src in case plane src changed but that can be changed in future to return damage.

For the case when plane is not visible or plane update should not happen the first call to *iter_next* will return false. Note that this helper use clipped *drm_plane_state.src*, so driver calling this helper should have called *drm_atomic_helper_check_plane_state()* earlier.

```
bool drm_atomic_helper_damage_iter_next(struct drm_atomic_helper_damage_iter *iter,  
                                         struct drm_rect *rect)
```

Advance the damage iterator.

Parameters

struct *drm_atomic_helper_damage_iter* *iter
The iterator to advance.

struct *drm_rect* *rect
Return a rectangle in fb coordinate clipped to plane src.

Description

Since plane src is in 16.16 fixed point and damage clips are whole number, this iterator round off clips that intersect with plane src. Round down for x1/y1 and round up for x2/y2 for the intersected coordinate. Similar rounding off for full plane src, in case it's returned as damage. This iterator will skip damage clips outside of plane src.

If the first call to iterator next returns false then it means no need to update the plane.

Return

True if the output is valid, false if reached the end.

```
bool drm_atomic_helper_damage_merged(const struct drm_plane_state *old_state, struct  
                                     drm_plane_state *state, struct drm_rect *rect)
```

Merged plane damage

Parameters

const struct drm_plane_state *old_state

Old plane state for validation.

struct drm_plane_state *state

Plane state from which to iterate the damage clips.

struct drm_rect *rect

Returns the merged damage rectangle

Description

This function merges any valid plane damage clips into one rectangle and returns it in **rect**.

For details see: [drm_atomic_helper_damage_iter_init\(\)](#) and [drm_atomic_helper_damage_iter_next\(\)](#).

Return

True if there is valid plane damage otherwise false.

drm_atomic_for_each_plane_damage

`drm_atomic_for_each_plane_damage (iter, rect)`

Iterator macro for plane damage.

Parameters

iter

The iterator to advance.

rect

Return a rectangle in fb coordinate clipped to plane src.

Description

Note that if the first call to iterator macro return false then no need to do plane update. Iterator will return full plane src when damage is not passed by user-space.

struct drm_atomic_helper_damage_iter

Closure structure for damage iterator.

Definition:

```
struct drm_atomic_helper_damage_iter {  
};
```

Members

Description

This structure tracks state needed to walk the list of plane damage clips.

4.10 Display Modes Function Reference

enum **drm_mode_status**

hardware support status of a mode

Constants

MODE_OK

Mode OK

MODE_HSYNC

hsync out of range

MODE_VSYNC

vsync out of range

MODE_H_ILLEGAL

mode has illegal horizontal timings

MODE_V_ILLEGAL

mode has illegal vertical timings

MODE_BAD_WIDTH

requires an unsupported linepitch

MODE_NOMODE

no mode with a matching name

MODE_NO_INTERLACE

interlaced mode not supported

MODE_NO_DBLESCAN

doublescan mode not supported

MODE_NO_VSCAN

multiscan mode not supported

MODE_MEM

insufficient video memory

MODE_VIRTUAL_X

mode width too large for specified virtual size

MODE_VIRTUAL_Y

mode height too large for specified virtual size

MODE_MEM_VIRT

insufficient video memory given virtual size

MODE_NOCLOCK

no fixed clock available

MODE_CLOCK_HIGH

clock required is too high

MODE_CLOCK_LOW

clock required is too low

MODE_CLOCK_RANGE

clock/mode isn't in a ClockRange

MODE_BAD_HVALUE

horizontal timing was out of range

MODE_BAD_VVALUE

vertical timing was out of range

MODE_BAD_VSCAN

VScan value out of range

MODE_HSYNC_NARROW

horizontal sync too narrow

MODE_HSYNC_WIDE

horizontal sync too wide

MODE_HBLANK_NARROW

horizontal blanking too narrow

MODE_HBLANK_WIDE

horizontal blanking too wide

MODE_VSYNC_NARROW

vertical sync too narrow

MODE_VSYNC_WIDE

vertical sync too wide

MODE_VBLANK_NARROW

vertical blanking too narrow

MODE_VBLANK_WIDE

vertical blanking too wide

MODE_PANEL

exceeds panel dimensions

MODE_INTERLACE_WIDTH

width too large for interlaced mode

MODE_ONE_WIDTH

only one width is supported

MODE_ONE_HEIGHT

only one height is supported

MODE_ONE_SIZE

only one resolution is supported

MODE_NO_REDUCED

monitor doesn't accept reduced blanking

MODE_NO_STEREO

stereo modes not supported

MODE_NO_420

ycbcr 420 modes not supported

MODE_STALE

mode has become stale

MODE_BAD

unspecified reason

MODE_ERROR

error condition

Description

This enum is used to filter out modes not supported by the driver/hardware combination.

DRM_MODE_RES_MM

DRM_MODE_RES_MM (res, dpi)

Calculates the display size from resolution and DPI

Parameters**res**

The resolution in pixel

dpi

The number of dots per inch

DRM_MODE_INIT

DRM_MODE_INIT (hz, hd, vd, hd_mm, vd_mm)

Initialize display mode

Parameters**hz**

Vertical refresh rate in Hertz

hd

Horizontal resolution, width

vd

Vertical resolution, height

hd_mm

Display width in millimeters

vd_mm

Display height in millimeters

Description

This macro initializes a *drm_display_mode* that contains information about refresh rate, resolution and physical size.

DRM_SIMPLE_MODE

DRM_SIMPLE_MODE (hd, vd, hd_mm, vd_mm)

Simple display mode

Parameters**hd**

Horizontal resolution, width

vd

Vertical resolution, height

hd_mm

Display width in millimeters

vd_mm

Display height in millimeters

Description

This macro initializes a *drm_display_mode* that only contains info about resolution and physical size.

struct **drm_display_mode**

DRM kernel-internal display mode structure

Definition:

```
struct drm_display_mode {
    int clock;
    u16 hdisplay;
    u16 hsync_start;
    u16 hsync_end;
    u16 httotal;
    u16 hskew;
    u16 vdisplay;
    u16 vsync_start;
    u16 vsync_end;
    u16 vttotal;
    u16 vscan;
    u32 flags;
    int crtc_clock;
    u16 crtc_hdisplay;
    u16 crtc_hblank_start;
    u16 crtc_hblank_end;
    u16 crtc_hsync_start;
    u16 crtc_hsync_end;
    u16 crtc_httotal;
    u16 crtc_hskew;
    u16 crtc_vdisplay;
    u16 crtc_vblank_start;
    u16 crtc_vblank_end;
    u16 crtc_vsync_start;
    u16 crtc_vsync_end;
    u16 crtc_vttotal;
    u16 width_mm;
    u16 height_mm;
    u8 type;
    bool expose_to_userspace;
    struct list_head head;
    char name[DRM_DISPLAY_MODE_LEN];
    enum drm_mode_status status;
    enum hdmi_picture_aspect picture_aspect_ratio;
```

```
};
```

Members

clock

Pixel clock in kHz.

hdisplay

horizontal display size

hsync_start

horizontal sync start

hsync_end

horizontal sync end

htotal

horizontal total size

hskew

horizontal skew?!

vdisplay

vertical display size

vsync_start

vertical sync start

vsync_end

vertical sync end

vtotal

vertical total size

vscan

vertical scan?!

flags

Sync and timing flags:

- `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`: <deprecated>
- `DRM_MODE_FLAG_PIXMUX`: <deprecated>

- `DRM_MODE_FLAG_DBLCLK`: double-clocked mode.
- `DRM_MODE_FLAG_CLKDIV2`: half-clocked mode.

Additionally there's flags to specify how 3D modes are packed:

- `DRM_MODE_FLAG_3D_NONE`: normal, non-3D mode.
- `DRM_MODE_FLAG_3D_FRAME_PACKING`: 2 full frames for left and right.
- `DRM_MODE_FLAG_3D_FIELD_ALTERNATIVE`: interleaved like fields.
- `DRM_MODE_FLAG_3D_LINE_ALTERNATIVE`: interleaved lines.
- `DRM_MODE_FLAG_3D_SIDE_BY_SIDE_FULL`: side-by-side full frames.
- `DRM_MODE_FLAG_3D_L_DEPTH`: ?
- `DRM_MODE_FLAG_3D_L_DEPTH_GFX_GFX_DEPTH`: ?
- `DRM_MODE_FLAG_3D_TOP_AND_BOTTOM`: frame split into top and bottom parts.
- `DRM_MODE_FLAG_3D_SIDE_BY_SIDE_HALF`: frame split into left and right parts.

crtc_clock

Actual pixel or dot clock in the hardware. This differs from the logical **clock** when e.g. using interlacing, double-clocking, stereo modes or other fancy stuff that changes the timings and signals actually sent over the wire.

This is again in kHz.

Note that with digital outputs like HDMI or DP there's usually a massive confusion between the dot clock and the signal clock at the bit encoding level. Especially when a 8b/10b encoding is used and the difference is exactly a factor of 10.

crtc_hdisplay

hardware mode horizontal display size

crtc_hblank_start

hardware mode horizontal blank start

crtc_hblank_end

hardware mode horizontal blank end

crtc_hsync_start

hardware mode horizontal sync start

crtc_hsync_end

hardware mode horizontal sync end

crtc_htotal

hardware mode horizontal total size

crtc_hskew

hardware mode horizontal skew?!

crtc_vdisplay

hardware mode vertical display size

crtc_vblank_start

hardware mode vertical blank start

crtc_vblank_end

hardware mode vertical blank end

crtc_vsync_start

hardware mode vertical sync start

crtc_vsync_end

hardware mode vertical sync end

crtc_vtotal

hardware mode vertical total size

width_mm

Addressable size of the output in mm, projectors should set this to 0.

height_mm

Addressable size of the output in mm, projectors should set this to 0.

type

A bitmask of flags, mostly about the source of a mode. Possible flags are:

- `DRM_MODE_TYPE_PREFERRED`: Preferred mode, usually the native resolution of an LCD panel. There should only be one preferred mode per connector at any given time.
- `DRM_MODE_TYPE_DRIVER`: Mode created by the driver, which is all of them really. Drivers must set this bit for all modes they create and expose to userspace.
- `DRM_MODE_TYPE_USERDEF`: Mode defined or selected via the kernel command line.

Plus a big list of flags which shouldn't be used at all, but are still around since these flags are also used in the userspace ABI. We no longer accept modes with these types though:

- `DRM_MODE_TYPE_BUILTIN`: Meant for hard-coded modes, unused. Use `DRM_MODE_TYPE_DRIVER` instead.
- `DRM_MODE_TYPE_DEFAULT`: Again a leftover, use `DRM_MODE_TYPE_PREFERRED` instead.
- `DRM_MODE_TYPE_CLOCK_C` and `DRM_MODE_TYPE_CRTC_C`: Define leftovers which are stuck around for hysterical raisins only. No one has an idea what they were meant for. Don't use.

expose_to_userspace

Indicates whether the mode is to be exposed to the userspace. This is to maintain a set of exposed modes while preparing user-mode's list in `drm_mode_getconnector` ioctl. The purpose of this only lies in the ioctl function, and is not to be used outside the function.

head

struct list_head for mode lists.

name

Human-readable name of the mode, filled out with `drm_mode_set_name()`.

status

Status of the mode, used to filter out modes not supported by the hardware. See enum `drm_mode_status`.

picture_aspect_ratio

Field for setting the HDMI picture aspect ratio of a mode.

void **drm_mode_debug_printmodeline**(const struct *drm_display_mode* *mode)
print a mode to dmesg

Parameters

const struct drm_display_mode *mode
mode to print

Description

Describe **mode** using DRM_DEBUG.

struct *drm_display_mode* ***drm_mode_create**(struct *drm_device* *dev)
create a new display mode

Parameters

struct drm_device *dev
DRM device

Description

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

Return

Pointer to new mode on success, NULL on error.

void **drm_mode_destroy**(struct *drm_device* *dev, struct *drm_display_mode* *mode)
remove a mode

Parameters

struct drm_device *dev
DRM device

struct drm_display_mode *mode
mode to remove

Description

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

void **drm_mode_probed_add**(struct *drm_connector* *connector, struct *drm_display_mode* *mode)
add a mode to a connector's *probed_mode* list

Parameters

struct drm_connector *connector
connector the new mode

struct drm_display_mode *mode
mode data

Description

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

```
struct drm_display_mode *drm_analog_tv_mode(struct drm_device *dev, enum  
                                           drm_connector_tv_mode tv_mode, unsigned  
                                           long pixel_clock_hz, unsigned int hdisplay,  
                                           unsigned int vdisplay, bool interlace)
```

create a display mode for an analog TV

Parameters

struct *drm_device* *dev
drm device

enum *drm_connector_tv_mode* tv_mode
TV Mode standard to create a mode for. See `DRM_MODE_TV_MODE_*`.

unsigned long pixel_clock_hz
Pixel Clock Frequency, in Hertz

unsigned int hdisplay
hdisplay size

unsigned int vdisplay
vdisplay size

bool interlace
whether to compute an interlaced mode

Description

This function creates a *struct *drm_display_mode** instance suited for an analog TV output, for one of the usual analog TV mode.

Note that **hdisplay** is larger than the usual constraints for the PAL and NTSC timings, and we'll choose to ignore most timings constraints to reach those resolutions.

A pointer to the mode, allocated with *drm_mode_create()*. Returns NULL on error.

Return

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

create a modeline based on the CVT algorithm

Parameters

struct *drm_device* *dev
drm device

int hdisplay
hdisplay size

int vdisplay
vdisplay size

int vrefresh
vrefresh rate

bool reduced
whether to use reduced blanking

bool interlaced

whether to compute an interlaced mode

bool margins

whether to add margins (borders)

Description

This function is called to generate the modeline based on CVT algorithm according to the hdisplay, vdisplay, vrefresh. It is based from the VESA(TM) Coordinated Video Timing Generator by Graham Loveridge April 9, 2003 available at <http://www.elo.utfsm.cl/~elo212/docs/CVTd6r1.xls>

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

Return

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

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

create the modeline based on the full GTF algorithm

Parameters

struct *drm_device* *dev
drm device

int hdisplay
hdisplay size

int vdisplay
vdisplay size

int vrefresh
vrefresh rate.

bool interlaced
whether to compute an interlaced mode

int margins
desired margin (borders) size

int GTF_M
extended GTF formula parameters

int GTF_2C
extended GTF formula parameters

int GTF_K
extended GTF formula parameters

int GTF_2J
extended GTF formula parameters

Description

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

Return

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

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

create the modeline based on the GTF algorithm

Parameters

struct drm_device *dev
drm device

int hdisplay
hdisplay size

int vdisplay
vdisplay size

int vrefresh
vrefresh rate.

bool interlaced
whether to compute an interlaced mode

int margins
desired margin (borders) size

Description

return the modeline based on GTF algorithm

This function is to create the modeline based on the GTF algorithm. Generalized Timing Formula is derived from:

GTF Spreadsheet by Andy Morrish (1/5/97) available at <https://www.vesa.org>

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

Standard GTF parameters:

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

Return

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

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

fill in **dmode** using **vm**,

Parameters

const struct videomode *vm

videomode structure to use as source

struct drm_display_mode *dmode

drm_display_mode structure to use as destination

Description

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

void **drm_display_mode_to_videomode**(const struct *drm_display_mode* *dmode, struct videomode *vm)

fill in **vm** using **dmode**,

Parameters

const struct drm_display_mode *dmode

drm_display_mode structure to use as source

struct videomode *vm

videomode structure to use as destination

Description

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

void **drm_bus_flags_from_videomode**(const struct videomode *vm, u32 *bus_flags)

extract information about pixelclk and DE polarity from videomode and store it in a separate variable

Parameters

const struct videomode *vm

videomode structure to use

u32 *bus_flags

information about pixelclk, sync and DE polarity will be stored here

Description

Sets DRM_BUS_FLAG_DE_(LOW|HIGH), DRM_BUS_FLAG_PIXDATA_DRIVE_(POS|NEG)EDGE and DISPLAY_FLAGS_SYNC_(POS|NEG)EDGE in **bus_flags** according to DISPLAY_FLAGS found in **vm**

int **of_get_drm_display_mode**(struct device_node *np, struct *drm_display_mode* *dmode, u32 *bus_flags, int index)

get a drm_display_mode from devicetree

Parameters

struct device_node *np

device_node with the timing specification

struct drm_display_mode *dmode

will be set to the return value

u32 *bus_flags

information about pixelclk, sync and DE polarity

int index

index into the list of display timings in devicetree

Description

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

Return

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

int of_get_drm_panel_display_mode(struct device_node *np, struct *drm_display_mode* *dmode, u32 *bus_flags)

get a panel-timing `drm_display_mode` from devicetree

Parameters

struct device_node *np

device_node with the panel-timing specification

struct drm_display_mode *dmode

will be set to the return value

u32 *bus_flags

information about pixelclk, sync and DE polarity

Description

The mandatory Device Tree properties width-mm and height-mm are read and set on the display mode.

Return

Zero on success, negative error code on failure.

void drm_mode_set_name(struct *drm_display_mode* *mode)

set the name on a mode

Parameters

struct drm_display_mode *mode

name will be set in this mode

Description

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

int drm_mode_vrefresh(const struct *drm_display_mode* *mode)

get the vrefresh of a mode

Parameters

const struct drm_display_mode *mode

mode

Return

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

```
void drm_mode_get_hv_timing(const struct drm_display_mode *mode, int *hdisplay, int
                             *vdisplay)
```

Fetches hdisplay/vdisplay for given mode

Parameters

const struct *drm_display_mode* *mode
mode to query

int *hdisplay
hdisplay value to fill in

int *vdisplay
vdisplay value to fill in

Description

The vdisplay value will be doubled if the specified mode is a stereo mode of the appropriate layout.

```
void drm_mode_set_crtcinfo(struct drm_display_mode *p, int adjust_flags)
    set CRTC modesetting timing parameters
```

Parameters

struct *drm_display_mode* *p
mode

int adjust_flags
a combination of adjustment flags

Description

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

- The CRTC_INTERLACE_HALVE_V flag can be used to halve vertical timings of interlaced modes.
- The CRTC_STEREO_DOUBLE flag can be used to compute the timings for buffers containing two eyes (only adjust the timings when needed, eg. for “frame packing” or “side by side full”).
- The CRTC_NO_DBLSCAN and CRTC_NO_VSCAN flags request that adjustment *not* be performed for doublescan and vscan > 1 modes respectively.

```
void drm_mode_copy(struct drm_display_mode *dst, const struct drm_display_mode *src)
    copy the mode
```

Parameters

struct *drm_display_mode* *dst
mode to overwrite

const struct *drm_display_mode* *src
mode to copy

Description

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

void **drm_mode_init**(struct *drm_display_mode* *dst, const struct *drm_display_mode* *src)
initialize the mode from another mode

Parameters

struct drm_display_mode *dst
mode to overwrite

const struct drm_display_mode *src
mode to copy

Description

Copy an existing mode into another mode, zeroing the list head of the destination mode. Typically used to guarantee the list head is not left with stack garbage in on-stack modes.

struct *drm_display_mode* ***drm_mode_duplicate**(struct *drm_device* *dev, const struct *drm_display_mode* *mode)
allocate and duplicate an existing mode

Parameters

struct drm_device *dev
drm_device to allocate the duplicated mode for

const struct drm_display_mode *mode
mode to duplicate

Description

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

Return

Pointer to duplicated mode on success, NULL on error.

bool **drm_mode_match**(const struct *drm_display_mode* *mode1, const struct *drm_display_mode* *mode2, unsigned int match_flags)
test modes for (partial) equality

Parameters

const struct drm_display_mode *mode1
first mode

const struct drm_display_mode *mode2
second mode

unsigned int match_flags
which parts need to match (DRM_MODE_MATCH_*)

Description

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

Return

True if the modes are (partially) equal, false otherwise.

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

test modes for equality

Parameters

const struct *drm_display_mode* *mode1

first mode

const struct *drm_display_mode* *mode2

second mode

Description

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

Return

True if the modes are equal, false otherwise.

bool **drm_mode_equal_no_clocks**(const struct *drm_display_mode* *mode1, const struct *drm_display_mode* *mode2)

test modes for equality

Parameters

const struct *drm_display_mode* *mode1

first mode

const struct *drm_display_mode* *mode2

second mode

Description

Check to see if **mode1** and **mode2** are equivalent, but don't check the pixel clocks.

Return

True if the modes are equal, false otherwise.

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

test modes for equality

Parameters

const struct *drm_display_mode* *mode1

first mode

const struct *drm_display_mode* *mode2

second mode

Description

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

Return

True if the modes are equal, false otherwise.

```
enum drm_mode_status drm_mode_validate_driver(struct drm_device *dev, const struct  
                                              drm_display_mode *mode)
```

make sure the mode is somewhat sane

Parameters

```
struct drm_device *dev  
    drm device
```

```
const struct drm_display_mode *mode  
    mode to check
```

Description

First do basic validation on the mode, and then allow the driver to check for device/driver specific limitations via the optional *drm_mode_config_helper_funcs.mode_valid* hook.

Return

The mode status

```
enum drm_mode_status drm_mode_validate_size(const struct drm_display_mode *mode, int  
                                              maxX, int maxY)
```

make sure modes adhere to size constraints

Parameters

```
const struct drm_display_mode *mode  
    mode to check
```

```
int maxX  
    maximum width
```

```
int maxY  
    maximum height
```

Description

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

Return

The mode status

```
enum drm_mode_status drm_mode_validate_ycbcr420(const struct drm_display_mode  
                                                  *mode, struct drm_connector  
                                                  *connector)
```

add 'ycbcr420-only' modes only when allowed

Parameters

```
const struct drm_display_mode *mode  
    mode to check
```

```
struct drm_connector *connector  
    drm connector under action
```

Description

This function is a helper which can be used to filter out any YCBCR420 only mode, when the source doesn't support it.

Return

The mode status

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

remove invalid modes from mode list

Parameters

struct drm_device *dev

DRM device

struct list_head *mode_list

list of modes to check

bool verbose

be verbose about it

Description

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

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

sort mode list

Parameters

struct list_head *mode_list

list of `drm_display_mode` structures to sort

Description

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

void **drm_connector_list_update**(struct *drm_connector* *connector)

update the mode list for the connector

Parameters

struct drm_connector *connector

the connector to update

Description

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

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

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

parse command line modeline for connector

Parameters

const char *mode_option
optional per connector mode option

const struct drm_connector *connector
connector to parse modeline for

struct drm_cmdline_mode *mode
preallocated `drm_cmdline_mode` structure to fill out

Description

This parses **mode_option** command line modeline for modes and options to configure the connector.

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

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

Additional options can be provided following the mode, using a comma to separate each option. Valid options can be found in `Documentation/fb/modedb.rst`.

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

Return

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

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

convert a command line modeline into a DRM display mode

Parameters

struct drm_device *dev
DRM device to create the new mode for

struct drm_cmdline_mode *cmd
input command line modeline

Return

Pointer to converted mode on success, NULL on error.

bool **drm_mode_is_420_only**(const struct *drm_display_info* *display, const struct *drm_display_mode* *mode)

if a given videomode can be only supported in YCBCR420 output format

Parameters

const struct drm_display_info *display
display under action

const struct drm_display_mode *mode
video mode to be tested.

Return

true if the mode can be supported in YCBCR420 format false if not.

bool **drm_mode_is_420_also**(const struct *drm_display_info* *display, const struct *drm_display_mode* *mode)

if a given videomode can be supported in YCBCR420 output format also (along with RGB/YCBCR444/422)

Parameters

const struct *drm_display_info* *display
display under action.

const struct *drm_display_mode* *mode
video mode to be tested.

Return

true if the mode can be support YCBCR420 format false if not.

bool **drm_mode_is_420**(const struct *drm_display_info* *display, const struct *drm_display_mode* *mode)

if a given videomode can be supported in YCBCR420 output format

Parameters

const struct *drm_display_info* *display
display under action.

const struct *drm_display_mode* *mode
video mode to be tested.

Return

true if the mode can be supported in YCBCR420 format false if not.

4.11 Connector Abstraction

In DRM connectors are the general abstraction for display sinks, and include also fixed panels or anything else that can display pixels in some form. As opposed to all other KMS objects representing hardware (like CRTC, encoder or plane abstractions) connectors can be hotplugged and unplugged at runtime. Hence they are reference-counted using *drm_connector_get()* and *drm_connector_put()*.

KMS driver must create, initialize, register and attach at a *struct drm_connector* for each such sink. The instance is created as other KMS objects and initialized by setting the following fields. The connector is initialized with a call to *drm_connector_init()* with a pointer to the *struct drm_connector_funcs* and a connector type, and then exposed to userspace with a call to *drm_connector_register()*.

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_connector_attach_encoder()*. The driver must also set the *drm_connector.encoder* field to point to the attached encoder.

For connectors which are not fixed (like built-in panels) the driver needs to support hotplug notifications. The simplest way to do that is by using the probe helpers, see `drm_kms_helper_poll_init()` for connectors which don't have hardware support for hotplug interrupts. Connectors with hardware hotplug support can instead use e.g. `drm_helper_hpd_irq_event()`.

4.11.1 Connector Functions Reference

enum **drm_connector_status**
status for a *drm_connector*

Constants

connector_status_connected
The connector is definitely connected to a sink device, and can be enabled.

connector_status_disconnected
The connector isn't connected to a sink device which can be autodetect. For digital outputs like DP or HDMI (which can be reliably probed) this means there's really nothing there. It is driver-dependent whether a connector with this status can be lit up or not.

connector_status_unknown
The connector's status could not be reliably detected. This happens when probing would either cause flicker (like load-detection when the connector is in use), or when a hardware resource isn't available (like when load-detection needs a free CRTC). It should be possible to light up the connector with one of the listed fallback modes. For default configuration userspace should only try to light up connectors with unknown status when there's not connector with **connector_status_connected**.

Description

This enum is used to track the connector status. There are no separate #defines for the uapi!

enum **drm_connector_registration_state**
userspace registration status for a *drm_connector*

Constants

DRM_CONNECTOR_INITIALIZING
The connector has just been created, but has yet to be exposed to userspace. There should be no additional restrictions to how the state of this connector may be modified.

DRM_CONNECTOR_REGISTERED
The connector has been fully initialized and registered with sysfs, as such it has been exposed to userspace. There should be no additional restrictions to how the state of this connector may be modified.

DRM_CONNECTOR_UNREGISTERED
The connector has either been exposed to userspace and has since been unregistered and removed from userspace, or the connector was unregistered before it had a chance to be exposed to userspace (e.g. still in the **DRM_CONNECTOR_INITIALIZING** state). When a connector is unregistered, there are additional restrictions to how its state may be modified:

- An unregistered connector may only have its DPMS changed from On->Off. Once DPMS is changed to Off, it may not be switched back to On.

- Modesets are not allowed on unregistered connectors, unless they would result in disabling its assigned CRTC. This means disabling a CRTC on an unregistered connector is OK, but enabling one is not.
- Removing a CRTC from an unregistered connector is OK, but new CRTCs may never be assigned to an unregistered connector.

Description

This enum is used to track the status of initializing a connector and registering it with userspace, so that DRM can prevent bogus modesets on connectors that no longer exist.

enum **drm_connector_tv_mode**

Analog TV output mode

Constants

DRM_MODE_TV_MODE_NTSC

CCIR System M (aka 525-lines) together with the NTSC Color Encoding.

DRM_MODE_TV_MODE_NTSC_443

Variant of **DRM_MODE_TV_MODE_NTSC**. Uses a color subcarrier frequency of 4.43 MHz.

DRM_MODE_TV_MODE_NTSC_J

Variant of **DRM_MODE_TV_MODE_NTSC** used in Japan. Uses a black level equals to the blanking level.

DRM_MODE_TV_MODE_PAL

CCIR System B together with the PAL color system.

DRM_MODE_TV_MODE_PAL_M

CCIR System M (aka 525-lines) together with the PAL color encoding

DRM_MODE_TV_MODE_PAL_N

CCIR System N together with the PAL color encoding. It uses 625 lines, but has a color subcarrier frequency of 3.58MHz, the SECAM color space, and narrower channels compared to most of the other PAL variants.

DRM_MODE_TV_MODE_SECAM

CCIR System B together with the SECAM color system.

DRM_MODE_TV_MODE_MAX

Number of analog TV output modes.

Internal implementation detail; this is not uABI.

Description

This enum is used to indicate the TV output mode used on an analog TV connector.

WARNING: The values of this enum is uABI since they're exposed in the "TV mode" connector property.

struct **drm_scrambling**

sink's scrambling support.

Definition:

```
struct drm_scrambling {
    bool supported;
    bool low_rates;
};
```

Members

supported

scrambling supported for rates > 340 Mhz.

low_rates

scrambling supported for rates <= 340 Mhz.

struct drm_hdmi_dsc_cap

DSC capabilities of HDMI sink

Definition:

```
struct drm_hdmi_dsc_cap {
    bool v_1p2;
    bool native_420;
    bool all_bpp;
    u8 bpc_supported;
    u8 max_slices;
    int clk_per_slice;
    u8 max_lanes;
    u8 max_frl_rate_per_lane;
    u8 total_chunk_kbytes;
};
```

Members

v_1p2

flag for dsc1.2 version support by sink

native_420

Does sink support DSC with 4:2:0 compression

all_bpp

Does sink support all bpp with 4:4:4: or 4:2:2 compressed formats

bpc_supported

compressed bpc supported by sink : 10, 12 or 16 bpc

max_slices

maximum number of Horizontal slices supported by

clk_per_slice

max pixel clock in MHz supported per slice

max_lanes

dsc max lanes supported for Fixed rate Link training

max_frl_rate_per_lane

maximum frl rate with DSC per lane

total_chunk_kbytes

max size of chunks in KBs supported per line

Description

Describes the DSC support provided by HDMI 2.1 sink. The information is fetched from additional HFVSDB blocks defined for HDMI 2.1.

struct `drm_hdmi_info`

runtime information about the connected HDMI sink

Definition:

```
struct drm_hdmi_info {
    struct drm_scdc scdc;
    unsigned long y420_vdb_modes[BITS_TO_LONGS(256)];
    unsigned long y420_cmbd_modes[BITS_TO_LONGS(256)];
    u8 y420_dc_modes;
    u8 max_frl_rate_per_lane;
    u8 max_lanes;
    struct drm_hdmi_dsc_cap dsc_cap;
};
```

Members**scdc**

sink's scdc support and capabilities

y420_vdb_modes

bitmap of modes which can support ycbcr420 output only (not normal RGB/YCBCR444/422 outputs). The max VIC defined by the CEA-861-G spec is 219, so the size is 256 bits to map up to 256 VICs.

y420_cmbd_modes

bitmap of modes which can support ycbcr420 output also, along with normal HDMI outputs. The max VIC defined by the CEA-861-G spec is 219, so the size is 256 bits to map up to 256 VICs.

y420_dc_modes

bitmap of deep color support index

max_frl_rate_per_lane

support fixed rate link

max_lanes

supported by sink

dsc_cap

DSC capabilities of the sink

Description

Describes if a given display supports advanced HDMI 2.0 features. This information is available in CEA-861-F extension blocks (like HF-VSDB).

enum `drm_link_status`

connector's link_status property value

Constants

DRM_LINK_STATUS_GOOD

DP Link is Good as a result of successful link training

DRM_LINK_STATUS_BAD

DP Link is BAD as a result of link training failure

Description

This enum is used as the connector's link status property value. It is set to the values defined in uapi.

enum **drm_panel_orientation**

panel_orientation info for *drm_display_info*

Constants

DRM_MODE_PANEL_ORIENTATION_UNKNOWN

The drm driver has not provided any panel orientation information (normal for non panels) in this case the "panel orientation" connector prop will not be attached.

DRM_MODE_PANEL_ORIENTATION_NORMAL

The top side of the panel matches the top side of the device's casing.

DRM_MODE_PANEL_ORIENTATION_BOTTOM_UP

The top side of the panel matches the bottom side of the device's casing, iow the panel is mounted upside-down.

DRM_MODE_PANEL_ORIENTATION_LEFT_UP

The left side of the panel matches the top side of the device's casing.

DRM_MODE_PANEL_ORIENTATION_RIGHT_UP

The right side of the panel matches the top side of the device's casing.

Description

This enum is used to track the (LCD) panel orientation. There are no separate #defines for the uapi!

struct **drm_monitor_range_info**

Panel's Monitor range in EDID for *drm_display_info*

Definition:

```
struct drm_monitor_range_info {
    u16 min_vfreq;
    u16 max_vfreq;
};
```

Members

min_vfreq

This is the min supported refresh rate in Hz from EDID's detailed monitor range.

max_vfreq

This is the max supported refresh rate in Hz from EDID's detailed monitor range

Description

This struct is used to store a frequency range supported by panel as parsed from EDID's detailed monitor range descriptor block.

struct `drm_luminance_range_info`

Panel's luminance range for `drm_display_info`. Calculated using data in EDID

Definition:

```
struct drm_luminance_range_info {
    u32 min_luminance;
    u32 max_luminance;
};
```

Members**`min_luminance`**

This is the min supported luminance value

`max_luminance`

This is the max supported luminance value

Description

This struct is used to store a luminance range supported by panel as calculated using data from EDID's static hdr metadata.

enum `drm_privacy_screen_status`

privacy screen status

Constants**`PRIVACY_SCREEN_DISABLED`**

The privacy-screen on the panel is disabled

`PRIVACY_SCREEN_ENABLED`

The privacy-screen on the panel is enabled

`PRIVACY_SCREEN_DISABLED_LOCKED`

The privacy-screen on the panel is disabled and locked (cannot be changed)

`PRIVACY_SCREEN_ENABLED_LOCKED`

The privacy-screen on the panel is enabled and locked (cannot be changed)

Description

This enum is used to track and control the state of the integrated privacy screen present on some display panels, via the "privacy-screen sw-state" and "privacy-screen hw-state" properties. Note the `_LOCKED` enum values are only valid for the "privacy-screen hw-state" property.

enum `drm_colorspace`

color space

Constants**`DRM_MODE_COLORIMETRY_DEFAULT`**

Driver specific behavior.

`DRM_MODE_COLORIMETRY_NO_DATA`

Driver specific behavior.

DRM_MODE_COLORIMETRY_SMPTE_170M_YCC

(HDMI) SMPTE ST 170M colorimetry format

DRM_MODE_COLORIMETRY_BT709_YCC

(HDMI, DP) ITU-R BT.709 colorimetry format

DRM_MODE_COLORIMETRY_XVYCC_601

(HDMI, DP) xvYCC601 colorimetry format

DRM_MODE_COLORIMETRY_XVYCC_709

(HDMI, DP) xvYCC709 colorimetry format

DRM_MODE_COLORIMETRY_SYCC_601

(HDMI, DP) sYCC601 colorimetry format

DRM_MODE_COLORIMETRY_OPYCC_601

(HDMI, DP) opYCC601 colorimetry format

DRM_MODE_COLORIMETRY_OPRGB

(HDMI, DP) opRGB colorimetry format

DRM_MODE_COLORIMETRY_BT2020_CYCC

(HDMI, DP) ITU-R BT.2020 Y'c C'bc C'rc (constant luminance) colorimetry format

DRM_MODE_COLORIMETRY_BT2020_RGB

(HDMI, DP) ITU-R BT.2020 R' G' B' colorimetry format

DRM_MODE_COLORIMETRY_BT2020_YCC

(HDMI, DP) ITU-R BT.2020 Y' C'b C'r colorimetry format

DRM_MODE_COLORIMETRY_DCI_P3_RGB_D65

(HDMI) SMPTE ST 2113 P3D65 colorimetry format

DRM_MODE_COLORIMETRY_DCI_P3_RGB_THEATER

(HDMI) SMPTE ST 2113 P3DCI colorimetry format

DRM_MODE_COLORIMETRY_RGB_WIDE_FIXED

(DP) RGB wide gamut fixed point colorimetry format

DRM_MODE_COLORIMETRY_RGB_WIDE_FLOAT

(DP) RGB wide gamut floating point (scRGB (IEC 61966-2-2)) colorimetry format

DRM_MODE_COLORIMETRY_BT601_YCC

(DP) ITU-R BT.601 colorimetry format The DP spec does not say whether this is the 525 or the 625 line version.

DRM_MODE_COLORIMETRY_COUNT

undescribed

Description

This enum is a consolidated colorimetry list supported by HDMI and DP protocol standard. The respective connectors will register a property with the subset of this list (supported by that respective protocol). Userspace will set the colorspace through a colorspace property which will be created and exposed to userspace.

DP definitions come from the DP v2.0 spec HDMI definitions come from the CTA-861-H spec

A note on YCC and RGB variants:

Since userspace is not aware of the encoding on the wire (RGB or YCbCr), drivers are free to pick the appropriate variant, regardless of what userspace selects. E.g., if BT2020_RGB is selected by userspace a driver will pick BT2020_YCC if the encoding on the wire is YUV444 or YUV420.

enum **drm_bus_flags**

bus_flags info for *drm_display_info*

Constants

DRM_BUS_FLAG_DE_LOW

The Data Enable signal is active low

DRM_BUS_FLAG_DE_HIGH

The Data Enable signal is active high

DRM_BUS_FLAG_PIXDATA_DRIVE_POSEDGE

Data is driven on the rising edge of the pixel clock

DRM_BUS_FLAG_PIXDATA_DRIVE_NEGEDGE

Data is driven on the falling edge of the pixel clock

DRM_BUS_FLAG_PIXDATA_SAMPLE_POSEDGE

Data is sampled on the rising edge of the pixel clock

DRM_BUS_FLAG_PIXDATA_SAMPLE_NEGEDGE

Data is sampled on the falling edge of the pixel clock

DRM_BUS_FLAG_DATA_MSB_TO_LSB

Data is transmitted MSB to LSB on the bus

DRM_BUS_FLAG_DATA_LSB_TO_MSB

Data is transmitted LSB to MSB on the bus

DRM_BUS_FLAG_SYNC_DRIVE_POSEDGE

Sync signals are driven on the rising edge of the pixel clock

DRM_BUS_FLAG_SYNC_DRIVE_NEGEDGE

Sync signals are driven on the falling edge of the pixel clock

DRM_BUS_FLAG_SYNC_SAMPLE_POSEDGE

Sync signals are sampled on the rising edge of the pixel clock

DRM_BUS_FLAG_SYNC_SAMPLE_NEGEDGE

Sync signals are sampled on the falling edge of the pixel clock

DRM_BUS_FLAG_SHARP_SIGNALS

Set if the Sharp-specific signals (SPL, CLS, PS, REV) must be used

Description

This enum defines signal polarities and clock edge information for signals on a bus as bitmask flags.

The clock edge information is conveyed by two sets of symbols, `DRM_BUS_FLAGS_*_DRIVE_*` and `DRM_BUS_FLAGS_*_SAMPLE_*`. When this enum is used to describe a bus from the point of view of the transmitter, the `*_DRIVE_*` flags should be used. When used from the point of view of the receiver, the `*_SAMPLE_*` flags should be used. The `*_DRIVE_*` and `*_SAMPLE_*` flags alias each other, with the `*_SAMPLE_POSEDGE` and `*_SAMPLE_NEGEDGE` flags being equal to `*_DRIVE_NEGEDGE` and `*_DRIVE_POSEDGE` respectively. This simplifies code as signals are usually sampled on the opposite edge of the driving edge. Transmitters and receivers may however need to take other signal timings into account to convert between driving and sample edges.

struct **drm_display_info**

runtime data about the connected sink

Definition:

```
struct drm_display_info {
    unsigned int width_mm;
    unsigned int height_mm;
    unsigned int bpc;
    enum subpixel_order subpixel_order;
#define DRM_COLOR_FORMAT_RGB444      (1<<0);
#define DRM_COLOR_FORMAT_YCBCR444   (1<<1);
#define DRM_COLOR_FORMAT_YCBCR422   (1<<2);
#define DRM_COLOR_FORMAT_YCBCR420   (1<<3);
    int panel_orientation;
    u32 color_formats;
    const u32 *bus_formats;
    unsigned int num_bus_formats;
    u32 bus_flags;
    int max_tmds_clock;
    bool dvi_dual;
    bool is_hdmi;
    bool has_audio;
    bool has_hdmi_infoframe;
    bool rgb_quant_range_selectable;
    u8 edid_hdmi_rgb444_dc_modes;
    u8 edid_hdmi_ycbcr444_dc_modes;
    u8 cea_rev;
    struct drm_hdmi_info hdmi;
    bool non_desktop;
    struct drm_monitor_range_info monitor_range;
    struct drm_luminance_range_info luminance_range;
    u8 mso_stream_count;
    u8 mso_pixel_overlap;
    u32 max_dsc_bpp;
    u8 *vics;
    int vics_len;
    u32 quirks;
};
```

```
};
```

Members

width_mm

Physical width in mm.

height_mm

Physical height in mm.

bpc

Maximum bits per color channel. Used by HDMI and DP outputs.

subpixel_order

Subpixel order of LCD panels.

panel_orientation

Read only connector property for built-in panels, indicating the orientation of the panel vs the device's casing. [drm_connector_init\(\)](#) sets this to `DRM_MODE_PANEL_ORIENTATION_UNKNOWN`. When not `UNKNOWN` this gets used by the `drm_fb_helpers` to rotate the fb to compensate and gets exported as `prop` to userspace.

color_formats

HDMI Color formats, selects between RGB and YCrCb modes. Used `DRM_COLOR_FORMAT_` defines, which are *not* the same ones as used to describe the pixel format in framebuffers, and also don't match the formats in **bus_formats** which are shared with v4l.

bus_formats

Pixel data format on the wire, somewhat redundant with **color_formats**. Array of size **num_bus_formats** encoded using `MEDIA_BUS_FMT_` defines shared with v4l and media drivers.

num_bus_formats

Size of **bus_formats** array.

bus_flags

Additional information (like pixel signal polarity) for the pixel data on the bus, using [enum drm_bus_flags](#) values `DRM_BUS_FLAGS_`.

max_tmds_clock

Maximum TMDS clock rate supported by the sink in kHz. 0 means undefined.

dvi_dual

Dual-link DVI sink?

is_hdmi

True if the sink is an HDMI device.

This field shall be used instead of calling [drm_detect_hdmi_monitor\(\)](#) when possible.

has_audio

True if the sink supports audio.

This field shall be used instead of calling [drm_detect_monitor_audio\(\)](#) when possible.

has_hdmi_infoframe

Does the sink support the HDMI infoframe?

rgb_quant_range_selectable

Does the sink support selecting the RGB quantization range?

edid_hdmi_rgb444_dc_modes

Mask of supported hdmi deep color modes in RGB 4:4:4. Even more stuff redundant with **bus_formats**.

edid_hdmi_ycbcr444_dc_modes

Mask of supported hdmi deep color modes in YCbCr 4:4:4. Even more stuff redundant with **bus_formats**.

cea_rev

CEA revision of the HDMI sink.

hdmi

advance features of a HDMI sink.

non_desktop

Non desktop display (HMD).

monitor_range

Frequency range supported by monitor range descriptor

luminance_range

Luminance range supported by panel

mso_stream_count

eDP Multi-SST Operation (MSO) stream count from the DisplayID VESA vendor block. 0 for conventional Single-Stream Transport (SST), or 2 or 4 MSO streams.

mso_pixel_overlap

eDP MSO segment pixel overlap, 0-8 pixels.

max_dsc_bpp

Maximum DSC target bitrate, if it is set to 0 the monitor's default value is used instead.

vics

Array of vics_len VICs. Internal to EDID parsing.

vics_len

Number of elements in vics. Internal to EDID parsing.

quirks

EDID based quirks. Internal to EDID parsing.

Description

Describes a given display (e.g. CRT or flat panel) and its limitations. For fixed display sinks like built-in panels there's not much difference between this and *struct drm_connector*. But for sinks with a real cable this structure is meant to describe all the things at the other end of the cable.

For sinks which provide an EDID this can be filled out by calling *drm_add_edid_modes()*.

struct drm_connector_tv_margins

TV connector related margins

Definition:

```
struct drm_connector_tv_margins {
    unsigned int bottom;
    unsigned int left;
    unsigned int right;
    unsigned int top;
};
```

Members

bottom

Bottom margin in pixels.

left

Left margin in pixels.

right

Right margin in pixels.

top

Top margin in pixels.

Description

Describes the margins in pixels to put around the image on TV connectors to deal with overscan.

struct **drm_tv_connector_state**

TV connector related states

Definition:

```
struct drm_tv_connector_state {
    enum drm_mode_subconnector select_subconnector;
    enum drm_mode_subconnector subconnector;
    struct drm_connector_tv_margins margins;
    unsigned int legacy_mode;
    unsigned int mode;
    unsigned int brightness;
    unsigned int contrast;
    unsigned int flicker_reduction;
    unsigned int overscan;
    unsigned int saturation;
    unsigned int hue;
};
```

Members

select_subconnector

selected subconnector

subconnector

detected subconnector

margins

TV margins

legacy_mode

Legacy TV mode, driver specific value

mode

TV mode

brightness

brightness in percent

contrast

contrast in percent

flicker_reduction

flicker reduction in percent

overscan

overscan in percent

saturation

saturation in percent

hue

hue in percent

struct `drm_connector_state`

mutable connector state

Definition:

```
struct drm_connector_state {
    struct drm_connector *connector;
    struct drm_crtc *crtc;
    struct drm_encoder *best_encoder;
    enum drm_link_status link_status;
    struct drm_atomic_state *state;
    struct drm_crtc_commit *commit;
    struct drm_tv_connector_state tv;
    bool self_refresh_aware;
    enum hdmi_picture_aspect picture_aspect_ratio;
    unsigned int content_type;
    unsigned int hdcp_content_type;
    unsigned int scaling_mode;
    unsigned int content_protection;
    enum drm_colorspace colorspace;
    struct drm_writeback_job *writeback_job;
    u8 max_requested_bpc;
    u8 max_bpc;
    enum drm_privacy_screen_status privacy_screen_sw_state;
    struct drm_property_blob *hdr_output_metadata;
};
```

Members**connector**

backpointer to the connector

crtc

CRTC to connect connector to, NULL if disabled.

Do not change this directly, use `drm_atomic_set_crtc_for_connector()` instead.

best_encoder

Used by the atomic helpers to select the encoder, through the `drm_connector_helper_funcs.atomic_best_encoder` or `drm_connector_helper_funcs.best_encoder` callbacks.

This is also used in the atomic helpers to map encoders to their current and previous connectors, see `drm_atomic_get_old_connector_for_encoder()` and `drm_atomic_get_new_connector_for_encoder()`.

NOTE: Atomic drivers must fill this out (either themselves or through helpers), for otherwise the GETCONNECTOR and GETENCODER IOCTLs will not return correct data to userspace.

link_status

Connector link_status to keep track of whether link is GOOD or BAD to notify userspace if retraining is necessary.

state

backpointer to global `drm_atomic_state`

commit

Tracks the pending commit to prevent use-after-free conditions.

Is only set when **crtc** is NULL.

tv

TV connector state

self_refresh_aware

This tracks whether a connector is aware of the self refresh state. It should be set to true for those connector implementations which understand the self refresh state. This is needed since the crtc registers the self refresh helpers and it doesn't know if the connectors downstream have implemented self refresh entry/exit.

Drivers should set this to true in `atomic_check` if they know how to handle self_refresh requests.

picture_aspect_ratio

Connector property to control the HDMI infoframe aspect ratio setting.

The `DRM_MODE_PICTURE_ASPECT_*` values much match the values for enum `hdmipicture_aspect`

content_type

Connector property to control the HDMI infoframe content type setting. The `DRM_MODE_CONTENT_TYPE_*` values much match the values.

hdcp_content_type

Connector property to pass the type of protected content. This is most commonly used for HDCP.

scaling_mode

Connector property to control the upscaling, mostly used for built-in panels.

content_protection

Connector property to request content protection. This is most commonly used for HDCP.

colorspace

State variable for Connector property to request colorspace change on Sink. This is most commonly used to switch to wider color gamuts like BT2020.

writeback_job

Writeback job for writeback connectors

Holds the framebuffer and out-fence for a writeback connector. As the writeback completion may be asynchronous to the normal commit cycle, the writeback job lifetime is managed separately from the normal atomic state by this object.

See also: [`drm_writeback_queue_job\(\)`](#) and [`drm_writeback_signal_completion\(\)`](#)

max_requested_bpc

Connector property to limit the maximum bit depth of the pixels.

max_bpc

Connector max_bpc based on the requested max_bpc property and the connector bpc limitations obtained from edid.

privacy_screen_sw_state

See [*Standard Connector Properties*](#)

hdr_output_metadata

DRM blob property for HDR output metadata

struct drm_connector_funcs

control connectors on a given device

Definition:

```
struct drm_connector_funcs {
    int (*dpms)(struct drm_connector *connector, int mode);
    void (*reset)(struct drm_connector *connector);
    enum drm_connector_status (*detect)(struct drm_connector *connector, bool
↪ force);
    void (*force)(struct drm_connector *connector);
    int (*fill_modes)(struct drm_connector *connector, uint32_t max_width,
↪ uint32_t max_height);
    int (*set_property)(struct drm_connector *connector, struct drm_property
↪ *property, uint64_t val);
    int (*late_register)(struct drm_connector *connector);
    void (*early_unregister)(struct drm_connector *connector);
    void (*destroy)(struct drm_connector *connector);
    struct drm_connector_state *(*atomic_duplicate_state)(struct drm_connector
↪ *connector);
    void (*atomic_destroy_state)(struct drm_connector *connector, struct drm_
↪ connector_state *state);
    int (*atomic_set_property)(struct drm_connector *connector, struct drm_
↪ connector_state *state, struct drm_property *property, uint64_t val);
    int (*atomic_get_property)(struct drm_connector *connector, const struct
↪ drm_connector_state *state, struct drm_property *property, uint64_t *val);
    void (*atomic_print_state)(struct drm_printer *p, const struct drm_
```

```

↪connector_state *state);
void (*oob_hotplug_event)(struct drm_connector *connector);
void (*debugfs_init)(struct drm_connector *connector, struct dentry *root);
};

```

Members

dpms

Legacy entry point to set the per-connector DPMS state. Legacy DPMS is exposed as a standard property on the connector, but diverted to this callback in the drm core. Note that atomic drivers don't implement the 4 level DPMS support on the connector any more, but instead only have an on/off "ACTIVE" property on the CRTC object.

This hook is not used by atomic drivers, remapping of the legacy DPMS property is entirely handled in the DRM core.

RETURNS:

0 on success or a negative error code on failure.

reset

Reset connector hardware and software state to off. This function isn't called by the core directly, only through `drm_mode_config_reset()`. It's not a helper hook only for historical reasons.

Atomic drivers can use `drm_atomic_helper_connector_reset()` to reset atomic state using this hook.

detect

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

This callback is optional, if not implemented the connector will be considered as always being attached.

FIXME:

Note that this hook is only called by the probe helper. It's not in the helper library vtable purely for historical reasons. The only DRM core entry point to probe connector state is **fill_modes**.

Note that the helper library will already hold `drm_mode_config.connection_mutex`. Drivers which need to grab additional locks to avoid races with concurrent modeset changes need to use `drm_connector_helper_funcs.detect_ctx` instead.

Also note that this callback can be called no matter the state the connector is in. Drivers that need the underlying device to be powered to perform the detection will first need to make sure it's been properly enabled.

RETURNS:

`drm_connector_status` indicating the connector's status.

force

This function is called to update internal encoder state when the connector is forced to a certain state by userspace, either through the sysfs interfaces or on the kernel cmdline. In that case the **detect** callback isn't called.

FIXME:

Note that this hook is only called by the probe helper. It's not in the helper library vtable purely for historical reasons. The only DRM core entry point to probe connector state is **fill_modes**.

fill_modes

Entry point for output detection and basic mode validation. The driver should reprobe the output if needed (e.g. when hotplug handling is unreliable), add all detected modes to *drm_connector.modes* and filter out any the device can't support in any configuration. It also needs to filter out any modes wider or higher than the parameters *max_width* and *max_height* indicate.

The drivers must also prune any modes no longer valid from *drm_connector.modes*. Furthermore it must update *drm_connector.status* and *drm_connector.edid*. If no EDID has been received for this output connector->edid must be NULL.

Drivers using the probe helpers should use *drm_helper_probe_single_connector_modes()* to implement this function.

RETURNS:

The number of modes detected and filled into *drm_connector.modes*.

set_property

This is the legacy entry point to update a property attached to the connector.

This callback is optional if the driver does not support any legacy driver-private properties. For atomic drivers it is not used because property handling is done entirely in the DRM core.

RETURNS:

0 on success or a negative error code on failure.

late_register

This optional hook can be used to register additional userspace interfaces attached to the connector, light backlight control, i2c, DP aux or similar interfaces. It is called late in the driver load sequence from *drm_connector_register()* when registering all the core drm connector interfaces. Everything added from this callback should be unregistered in the *early_unregister* callback.

This is called while holding *drm_connector.mutex*.

Returns:

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

early_unregister

This optional hook should be used to unregister the additional userspace interfaces attached to the connector from *late_register()*. It is called from *drm_connector_unregister()*, early in the driver unload sequence to disable userspace access before data structures are torndown.

This is called while holding *drm_connector.mutex*.

destroy

Clean up connector resources. This is called at driver unload time through *drm_mode_config_cleanup()*. It can also be called at runtime when a connector is being hot-unplugged for drivers that support connector hotplugging (e.g. DisplayPort MST).

atomic_duplicate_state

Duplicate the current atomic state for this connector and return it. The core and helpers guarantee that any atomic state duplicated with this hook and still owned by the caller (i.e. not transferred to the driver by calling `drm_mode_config_funcs.atomic_commit`) will be cleaned up by calling the **atomic_destroy_state** hook in this structure.

This callback is mandatory for atomic drivers.

Atomic drivers which don't subclass `struct drm_connector_state` should use `drm_atomic_helper_connector_duplicate_state()`. Drivers that subclass the state structure to extend it with driver-private state should use `__drm_atomic_helper_connector_duplicate_state()` to make sure shared state is duplicated in a consistent fashion across drivers.

It is an error to call this hook before `drm_connector.state` has been initialized correctly.

NOTE:

If the duplicate state references refcounted resources this hook must acquire a reference for each of them. The driver must release these references again in **atomic_destroy_state**.

RETURNS:

Duplicated atomic state or NULL when the allocation failed.

atomic_destroy_state

Destroy a state duplicated with **atomic_duplicate_state** and release or unreference all resources it references

This callback is mandatory for atomic drivers.

atomic_set_property

Decode a driver-private property value and store the decoded value into the passed-in state structure. Since the atomic core decodes all standardized properties (even for extensions beyond the core set of properties which might not be implemented by all drivers) this requires drivers to subclass the state structure.

Such driver-private properties should really only be implemented for truly hardware/vendor specific state. Instead it is preferred to standardize atomic extension and decode the properties used to expose such an extension in the core.

Do not call this function directly, use `drm_atomic_connector_set_property()` instead.

This callback is optional if the driver does not support any driver-private atomic properties.

NOTE:

This function is called in the state assembly phase of atomic modesets, which can be aborted for any reason (including on userspace's request to just check whether a configuration would be possible). Drivers MUST NOT touch any persistent state (hardware or software) or data structures except the passed in **state** parameter.

Also since userspace controls in which order properties are set this function must not do any input validation (since the state update is incomplete and hence likely inconsistent). Instead any such input validation must be done in the various `atomic_check` callbacks.

RETURNS:

0 if the property has been found, -EINVAL if the property isn't implemented by the driver (which shouldn't ever happen, the core only asks for properties attached to this connector). No other validation is allowed by the driver. The core already checks that the property value is within the range (integer, valid enum value, ...) the driver set when registering the property.

atomic_get_property

Reads out the decoded driver-private property. This is used to implement the GETCONNECTOR_IOCTL.

Do not call this function directly, use `drm_atomic_connector_get_property()` instead.

This callback is optional if the driver does not support any driver-private atomic properties.

RETURNS:

0 on success, -EINVAL if the property isn't implemented by the driver (which shouldn't ever happen, the core only asks for properties attached to this connector).

atomic_print_state

If driver subclasses `struct drm_connector_state`, it should implement this optional hook for printing additional driver specific state.

Do not call this directly, use `drm_atomic_connector_print_state()` instead.

oob_hotplug_event

This will get called when a hotplug-event for a drm-connector has been received from a source outside the display driver / device.

debugfs_init

Allows connectors to create connector-specific debugfs files.

Description

Each CRTC may have one or more connectors attached to it. The functions below allow the core DRM code to control connectors, enumerate available modes, etc.

struct drm_cmdline_mode

DRM Mode passed through the kernel command-line

Definition:

```
struct drm_cmdline_mode {
    char name[DRM_DISPLAY_MODE_LEN];
    bool specified;
    bool refresh_specified;
    bool bpp_specified;
    unsigned int pixel_clock;
    int xres;
    int yres;
    int bpp;
    int refresh;
    bool rb;
    bool interlace;
    bool cvt;
    bool margins;
    enum drm_connector_force force;
    unsigned int rotation_reflection;
```

```
enum drm_panel_orientation panel_orientation;
struct drm_connector_tv_margins tv_margins;
enum drm_connector_tv_mode tv_mode;
bool tv_mode_specified;
};
```

Members

name

Name of the mode.

specified

Has a mode been read from the command-line?

refresh_specified

Did the mode have a preferred refresh rate?

bpp_specified

Did the mode have a preferred BPP?

pixel_clock

Pixel Clock in kHz. Optional.

xres

Active resolution on the X axis, in pixels.

yres

Active resolution on the Y axis, in pixels.

bpp

Bits per pixels for the mode.

refresh

Refresh rate, in Hertz.

rb

Do we need to use reduced blanking?

interlace

The mode is interlaced.

cvt

The timings will be calculated using the VESA Coordinated Video Timings instead of looking up the mode from a table.

margins

Add margins to the mode calculation (1.8% of xres rounded down to 8 pixels and 1.8% of yres).

force

Ignore the hotplug state of the connector, and force its state to one of the `DRM_FORCE_*` values.

rotation_reflection

Initial rotation and reflection of the mode setup from the command line. See `DRM_MODE_ROTATE_*` and `DRM_MODE_REFLECT_*`. The only rotations supported are `DRM_MODE_ROTATE_0` and `DRM_MODE_ROTATE_180`.

panel_orientation

drm-connector "panel orientation" property override value,
DRM_MODE_PANEL_ORIENTATION_UNKNOWN if not set.

tv_margins

TV margins to apply to the mode.

tv_mode

TV mode standard. See DRM_MODE_TV_MODE_*.

tv_mode_specified

Did the mode have a preferred TV mode?

Description

Each connector can have an initial mode with additional options passed through the kernel command line. This structure allows to express those parameters and will be filled by the command-line parser.

struct drm_connector

central DRM connector control structure

Definition:

```
struct drm_connector {
    struct drm_device *dev;
    struct device *kdev;
    struct device_attribute *attr;
    struct fwnode_handle *fwnode;
    struct list_head head;
    struct list_head global_connector_list_entry;
    struct drm_mode_object base;
    char *name;
    struct mutex mutex;
    unsigned index;
    int connector_type;
    int connector_type_id;
    bool interlace_allowed;
    bool doublescan_allowed;
    bool stereo_allowed;
    bool ycbcr_420_allowed;
    enum drm_connector_registration_state registration_state;
    struct list_head modes;
    enum drm_connector_status status;
    struct list_head probed_modes;
    struct drm_display_info display_info;
    const struct drm_connector_funcs *funcs;
    struct drm_property_blob *edid_blob_ptr;
    struct drm_object_properties properties;
    struct drm_property *scaling_mode_property;
    struct drm_property *vrr_capable_property;
    struct drm_property *colorspace_property;
    struct drm_property_blob *path_blob_ptr;
    struct drm_property *max_bpc_property;
    struct drm_privacy_screen *privacy_screen;
```



```

    struct notifier_block privacy_screen_notifier;
    struct drm_property *privacy_screen_sw_state_property;
    struct drm_property *privacy_screen_hw_state_property;
#define DRM_CONNECTOR_POLL_HPD (1 << 0);
#define DRM_CONNECTOR_POLL_CONNECT (1 << 1);
#define DRM_CONNECTOR_POLL_DISCONNECT (1 << 2);
    uint8_t polled;
    int dpms;
    const struct drm_connector_helper_funcs *helper_private;
    struct drm_cmdline_mode cmdline_mode;
    enum drm_connector_force force;
    const struct drm_edid *edid_override;
    struct mutex edid_override_mutex;
    u64 epoch_counter;
    u32 possible_encoders;
    struct drm_encoder *encoder;
#define MAX_ELD_BYTES 128;
    uint8_t eld[MAX_ELD_BYTES];
    bool latency_present[2];
    int video_latency[2];
    int audio_latency[2];
    struct i2c_adapter *ddc;
    int null_edid_counter;
    unsigned bad_edid_counter;
    bool edid_corrupt;
    u8 real_edid_checksum;
    struct dentry *debugfs_entry;
    struct drm_connector_state *state;
    struct drm_property_blob *tile_blob_ptr;
    bool has_tile;
    struct drm_tile_group *tile_group;
    bool tile_is_single_monitor;
    uint8_t num_h_tile, num_v_tile;
    uint8_t tile_h_loc, tile_v_loc;
    uint16_t tile_h_size, tile_v_size;
    struct llist_node free_node;
    struct hdr_sink_metadata hdr_sink_metadata;
};

```

Members

dev

parent DRM device

kdev

kernel device for sysfs attributes

attr

sysfs attributes

fwnode

associated fwnode supplied by platform firmware

Drivers can set this to associate a fwnode with a connector, drivers are expected to get a reference on the fwnode when setting this. `drm_connector_cleanup()` will call `fwnode_handle_put()` on this.

head

List of all connectors on a **dev**, linked from `drm_mode_config.connector_list`. Protected by `drm_mode_config.connector_list_lock`, but please only use `drm_connector_list_iter` to walk this list.

global_connector_list_entry

Connector entry in the global connector-list, used by `drm_connector_find_by_fwnode()`.

base

base KMS object

name

human readable name, can be overwritten by the driver

mutex

Lock for general connector state, but currently only protects **registered**. Most of the connector state is still protected by `drm_mode_config.mutex`.

index

Compacted connector index, which matches the position inside the `mode_config.list` for drivers not supporting hot-add/removing. Can be used as an array index. It is invariant over the lifetime of the connector.

connector_type

one of the `DRM_MODE_CONNECTOR_<foo>` types from `drm_mode.h`

connector_type_id

index into connector type enum

interlace_allowed

Can this connector handle interlaced modes? Only used by `drm_helper_probe_single_connector_modes()` for mode filtering.

doublescan_allowed

Can this connector handle doublescan? Only used by `drm_helper_probe_single_connector_modes()` for mode filtering.

stereo_allowed

Can this connector handle stereo modes? Only used by `drm_helper_probe_single_connector_modes()` for mode filtering.

ycbcr_420_allowed

This bool indicates if this connector is capable of handling YCBCR 420 output. While parsing the EDID blocks it's very helpful to know if the source is capable of handling YCBCR 420 outputs.

registration_state

Is this connector initializing, exposed (registered) with userspace, or unregistered?

Protected by **mutex**.

modes

Modes available on this connector (from `fill_modes()` + user). Protected by `drm_mode_config.mutex`.

status

One of the `drm_connector_status` enums (connected, not, or unknown). Protected by `drm_mode_config.mutex`.

probed_modes

These are modes added by probing with DDC or the BIOS, before filtering is applied. Used by the probe helpers. Protected by `drm_mode_config.mutex`.

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 `drm_display_info.width_mm` and `drm_display_info.height_mm` fields with the physical size of the display.

Protected by `drm_mode_config.mutex`.

funcs

connector control functions

edid_blob_ptr

DRM property containing EDID if present. Protected by `drm_mode_config.mutex`. This should be updated only by calling `drm_connector_update_edid_property()`.

properties

property tracking for this connector

scaling_mode_property

Optional atomic property to control the upscaling. See `drm_connector_attach_content_protection_property()`.

vrr_capable_property

Optional property to help userspace query hardware support for variable refresh rate on a connector. Drivers can add the property to a connector by calling `drm_connector_attach_vrr_capable_property()`.

This should be updated only by calling `drm_connector_set_vrr_capable_property()`.

colorspace_property

Connector property to set the suitable colorspace supported by the sink.

path_blob_ptr

DRM blob property data for the DP MST path property. This should only be updated by calling `drm_connector_set_path_property()`.

max_bpc_property

Default connector property for the max bpc to be driven out of the connector.

privacy_screen

`drm_privacy_screen` for this connector, or NULL.

privacy_screen_notifier

privacy-screen notifier_block

privacy_screen_sw_state_property

Optional atomic property for the connector to control the integrated privacy screen.

privacy_screen_hw_state_property

Optional atomic property for the connector to report the actual integrated privacy screen state.

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, without causing flickering even when the connector is in use. DACs should rarely do this without a lot of testing.

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

dpms

Current dpms state. For legacy drivers the `drm_connector_funcs.dpms` callback must update this. For atomic drivers, this is handled by the core atomic code, and drivers must only take `drm_crtc_state.active` into account.

helper_private

mid-layer private data

cmdline_mode

mode line parsed from the kernel cmdline for this connector

force

a `DRM_FORCE_<foo>` state for forced mode sets

edid_override

Override EDID set via debugfs.

Do not modify or access outside of the `drm_edid_override_*` family of functions.

edid_override_mutex

Protect access to `edid_override`.

epoch_counter

used to detect any other changes in connector, besides status

possible_encoders

Bit mask of encoders that can drive this connector, `drm_encoder_index()` determines the index into the bitfield and the bits are set with `drm_connector_attach_encoder()`.

encoder

Currently bound encoder driving this connector, if any. Only really meaningful for non-atomic drivers. Atomic drivers should instead look at `drm_connector_state.best_encoder`, and in case they need the CRTC driving this output, `drm_connector_state.crtc`.

eld

EDID-like data, if present

latency_present

AV delay info from ELD, if found

video_latency

Video latency info from ELD, if found. [0]: progressive, [1]: interlaced

audio_latency

audio latency info from ELD, if found [0]: progressive, [1]: interlaced

ddc

associated ddc adapter. A connector usually has its associated ddc adapter. If a driver uses this field, then an appropriate symbolic link is created in connector sysfs directory to make it easy for the user to tell which i2c adapter is for a particular display.

The field should be set by calling `drm_connector_init_with_ddc()`.

null_edid_counter

track sinks that give us all zeros for the EDID. Needed to workaround some HW bugs where we get all 0s

bad_edid_counter

track sinks that give us an EDID with invalid checksum

edid_corrupt

Indicates whether the last read EDID was corrupt. Used in Displayport compliance testing - Displayport Link CTS Core 1.2 rev1.1 4.2.2.6

real_edid_checksum

real edid checksum for corrupted edid block. Required in Displayport 1.4 compliance testing rev1.1 4.2.2.6

debugfs_entry

debugfs directory for this connector

state

Current atomic state for this connector.

This is protected by `drm_mode_config.connection_mutex`. Note that nonblocking atomic commits access the current connector state without taking locks. Either by going through the `struct drm_atomic_state` pointers, see `for_each_oldnew_connector_in_state()`, `for_each_old_connector_in_state()` and `for_each_new_connector_in_state()`. Or through careful ordering of atomic commit operations as implemented in the atomic helpers, see `struct drm_crtc_commit`.

tile_blob_ptr

DRM blob property data for the tile property (used mostly by DP MST). This is meant for screens which are driven through separate display pipelines represented by `drm_crtc`, which might not be running with genlocked clocks. For tiled panels which are genlocked, like dual-link LVDS or dual-link DSI, the driver should try to not expose the tiling and virtualize both `drm_crtc` and `drm_plane` if needed.

This should only be updated by calling `drm_connector_set_tile_property()`.

has_tile

is this connector connected to a tiled monitor

tile_group

tile group for the connected monitor

tile_is_single_monitor

whether the tile is one monitor housing

num_h_tile

number of horizontal tiles in the tile group

num_v_tile

number of vertical tiles in the tile group

tile_h_loc

horizontal location of this tile

tile_v_loc

vertical location of this tile

tile_h_size

horizontal size of this tile.

tile_v_size

vertical size of this tile.

free_node

List used only by *drm_connector_list_iter* to be able to clean up a connector from any context, in conjunction with *drm_mode_config.connector_free_work*.

hdr_sink_metadata

HDR Metadata Information read from sink

Description

Each connector may be connected to one or more CRTC's, or may be clonable by another connector if they can share a CRTC. Each connector also has a specific position in the broader display (referred to as a 'screen' though it could span multiple monitors).

```
struct drm_connector *drm_connector_lookup(struct drm_device *dev, struct drm_file
                                           *file_priv, uint32_t id)
```

lookup connector object

Parameters

```
struct drm_device *dev
```

DRM device

```
struct drm_file *file_priv
```

drm file to check for lease against.

```
uint32_t id
```

connector object id

Description

This function looks up the connector object specified by id and takes a reference to it.

```
void drm_connector_get(struct drm_connector *connector)
```

acquire a connector reference

Parameters

```
struct drm_connector *connector
```

DRM connector

Description

This function increments the connector's refcount.

```
void drm_connector_put(struct drm_connector *connector)
```

release a connector reference

Parameters

struct drm_connector *connector
 DRM connector

Description

This function decrements the connector's reference count and frees the object if the reference count drops to zero.

bool **drm_connector_is_unregistered**(struct *drm_connector* *connector)
 has the connector been unregistered from userspace?

Parameters

struct drm_connector *connector
 DRM connector

Description

Checks whether or not **connector** has been unregistered from userspace.

Return

True if the connector was unregistered, false if the connector is registered or has not yet been registered with userspace.

struct **drm_tile_group**
 Tile group metadata

Definition:

```
struct drm_tile_group {
    struct kref refcount;
    struct drm_device *dev;
    int id;
    u8 group_data[8];
};
```

Members

refcount
 reference count

dev
 DRM device

id
 tile group id exposed to userspace

group_data
 Sink-private data identifying this group

Description

group_data corresponds to displayid vend/prod/serial for external screens with an EDID.

struct **drm_connector_list_iter**
 connector_list iterator

Definition:

```
struct drm_connector_list_iter {  
};
```

Members

Description

This iterator tracks state needed to be able to walk the `connector_list` within *struct `drm_mode_config`*. Only use together with *`drm_connector_list_iter_begin()`*, *`drm_connector_list_iter_end()`* and *`drm_connector_list_iter_next()`* respectively the convenience macro *`drm_for_each_connector_iter()`*.

Note that the return value of *`drm_connector_list_iter_next()`* is only valid up to the next *`drm_connector_list_iter_next()`* or *`drm_connector_list_iter_end()`* call. If you want to use the connector later, then you need to grab your own reference first using *`drm_connector_get()`*.

`drm_for_each_connector_iter`

`drm_for_each_connector_iter (connector, iter)`

connector_list iterator macro

Parameters

connector

struct `drm_connector` pointer used as cursor

iter

struct `drm_connector_list_iter`

Description

Note that **connector** is only valid within the list body, if you want to use **connector** after calling *`drm_connector_list_iter_end()`* then you need to grab your own reference first using *`drm_connector_get()`*.

`drm_connector_for_each_possible_encoder`

`drm_connector_for_each_possible_encoder (connector, encoder)`

iterate connector's possible encoders

Parameters

connector

struct `drm_connector` pointer

encoder

struct `drm_encoder` pointer used as cursor

`const char *drm_get_connector_type_name(unsigned int type)`

return a string for connector type

Parameters

unsigned int type

The connector type (`DRM_MODE_CONNECTOR_*`)

Return

the name of the connector type, or NULL if the type is not valid.

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

Init a preallocated connector

Parameters

```
struct drm_device *dev
    DRM device
```

```
struct drm_connector *connector
    the connector to init
```

```
const struct drm_connector_funcs *funcs
    callbacks for this connector
```

```
int connector_type
    user visible type of the connector
```

Description

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

At driver unload time the driver's *drm_connector_funcs.destroy* hook should call *drm_connector_cleanup()* and free the connector structure. The connector structure should not be allocated with *devm_kzalloc()*.

Note

consider using *drm_connector_init()* instead of *drm_connector_init()* to let the DRM managed resource infrastructure take care of cleanup and deallocation.

Return

Zero on success, error code on failure.

```
int drm_connector_init_with_ddc(struct drm_device *dev, struct drm_connector *connector,
                                const struct drm_connector_funcs *funcs, int
                                connector_type, struct i2c_adapter *ddc)
```

Init a preallocated connector

Parameters

```
struct drm_device *dev
    DRM device
```

```
struct drm_connector *connector
    the connector to init
```

```
const struct drm_connector_funcs *funcs
    callbacks for this connector
```

```
int connector_type
    user visible type of the connector
```

```
struct i2c_adapter *ddc
    pointer to the associated ddc adapter
```

Description

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

At driver unload time the driver's `drm_connector_funcs.destroy` hook should call `drm_connector_cleanup()` and free the connector structure. The connector structure should not be allocated with `devm_kzalloc()`.

Ensures that the `ddc` field of the connector is correctly set.

Note

consider using `drmm_connector_init()` instead of `drm_connector_init_with_ddc()` to let the DRM managed resource infrastructure take care of cleanup and deallocation.

Return

Zero on success, error code on failure.

```
int drmm_connector_init(struct drm_device *dev, struct drm_connector *connector, const
                        struct drm_connector_funcs *funcs, int connector_type, struct
                        i2c_adapter *ddc)
```

Init a preallocated connector

Parameters

struct drm_device *dev
DRM device

struct drm_connector *connector
the connector to init

const struct drm_connector_funcs *funcs
callbacks for this connector

int connector_type
user visible type of the connector

struct i2c_adapter *ddc
optional pointer to the associated ddc adapter

Description

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

Cleanup is automatically handled with a call to `drm_connector_cleanup()` in a DRM-managed action.

The connector structure should be allocated with `drmm_kzalloc()`.

Return

Zero on success, error code on failure.

```
void drm_connector_attach_edid_property(struct drm_connector *connector)
    attach edid property.
```

Parameters

struct drm_connector *connector
the connector

Description

Some connector types like `DRM_MODE_CONNECTOR_VIRTUAL` do not get a edid property attached by default. This function can be used to explicitly enable the edid property in these cases.

int **drm_connector_attach_encoder**(struct *drm_connector* *connector, struct *drm_encoder* *encoder)

attach a connector to an encoder

Parameters

struct drm_connector *connector
connector to attach

struct drm_encoder *encoder
encoder to attach **connector** to

Description

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

Return

Zero on success, negative errno on failure.

bool **drm_connector_has_possible_encoder**(struct *drm_connector* *connector, struct *drm_encoder* *encoder)

check if the connector and encoder are associated with each other

Parameters

struct drm_connector *connector
the connector

struct drm_encoder *encoder
the encoder

Return

True if **encoder** is one of the possible encoders for **connector**.

void **drm_connector_cleanup**(struct *drm_connector* *connector)
cleans up an initialised connector

Parameters

struct drm_connector *connector
connector to cleanup

Description

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

int **drm_connector_register**(struct *drm_connector* *connector)
register a connector

Parameters

struct drm_connector *connector
the connector to register

Description

Register userspace interfaces for a connector. Only call this for connectors which can be hot-plugged after *drm_dev_register()* has been called already, e.g. DP MST connectors. All other connectors will be registered automatically when calling *drm_dev_register()*.

When the connector is no longer available, callers must call *drm_connector_unregister()*.

Return

Zero on success, error code on failure.

void **drm_connector_unregister**(struct *drm_connector* *connector)
unregister a connector

Parameters

struct drm_connector *connector
the connector to unregister

Description

Unregister userspace interfaces for a connector. Only call this for connectors which have been registered explicitly by calling *drm_connector_register()*.

const char ***drm_get_connector_status_name**(enum *drm_connector_status* status)
return a string for connector status

Parameters

enum drm_connector_status status
connector status to compute name of

Description

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

Return

connector status string

void **drm_connector_list_iter_begin**(struct *drm_device* *dev, struct *drm_connector_list_iter* *iter)
initialize a connector_list iterator

Parameters

struct drm_device *dev
DRM device

struct drm_connector_list_iter *iter
connector_list iterator

Description

Sets **iter** up to walk the `drm_mode_config.connector_list` of **dev**. **iter** must always be cleaned up again by calling `drm_connector_list_iter_end()`. Iteration itself happens using `drm_connector_list_iter_next()` or `drm_for_each_connector_iter()`.

```
struct drm_connector *drm_connector_list_iter_next(struct drm_connector_list_iter *iter)
    return next connector
```

Parameters

```
struct drm_connector_list_iter *iter
    connector_list iterator
```

Return

the next connector for **iter**, or NULL when the list walk has completed.

```
void drm_connector_list_iter_end(struct drm_connector_list_iter *iter)
    tear down a connector_list iterator
```

Parameters

```
struct drm_connector_list_iter *iter
    connector_list iterator
```

Description

Tears down **iter** and releases any resources (like `drm_connector` references) acquired while walking the list. This must always be called, both when the iteration completes fully or when it was aborted without walking the entire list.

```
const char *drm_get_subpixel_order_name(enum subpixel_order order)
    return a string for a given subpixel enum
```

Parameters

```
enum subpixel_order order
    enum of subpixel_order
```

Description

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

Return

string describing an enumerated subpixel property

```
int drm_display_info_set_bus_formats(struct drm_display_info *info, const u32 *formats,
    unsigned int num_formats)
    set the supported bus formats
```

Parameters

```
struct drm_display_info *info
    display info to store bus formats in
```

```
const u32 *formats
    array containing the supported bus formats
```

unsigned int num_formats

the number of entries in the fmts array

Description

Store the supported bus formats in display info structure. See `MEDIA_BUS_FMT_*` definitions in `include/uapi/linux/media-bus-format.h` for a full list of available formats.

Return

0 on success or a negative error code on failure.

int **drm_get_tv_mode_from_name**(const char *name, size_t len)

Translates a TV mode name into its enum value

Parameters

const char *name

TV Mode name we want to convert

size_t len

Length of **name**

Description

Translates **name** into an *enum `drm_connector_tv_mode`*.

Return

the enum value on success, a negative errno otherwise.

int **drm_mode_create_dvi_i_properties**(struct *drm_device* *dev)

create DVI-I specific connector properties

Parameters

struct *drm_device* *dev

DRM device

Description

Called by a driver the first time a DVI-I connector is made.

Return

0

void **drm_connector_attach_dp_subconnector_property**(struct *drm_connector* *connector)

create subconnector property for DP

Parameters

struct *drm_connector* *connector

drm_connector to attach property

Description

Called by a driver when DP connector is created.

int **drm_connector_attach_content_type_property**(struct *drm_connector* *connector)

attach content-type property

Parameters

struct drm_connector *connector
connector to attach content type property on.

Description

Called by a driver the first time a HDMI connector is made.

Return

0

void **drm_connector_attach_tv_margin_properties**(struct *drm_connector* *connector)
attach TV connector margin properties

Parameters

struct drm_connector *connector
DRM connector

Description

Called by a driver when it needs to attach TV margin props to a connector. Typically used on SDTV and HDMI connectors.

int **drm_mode_create_tv_margin_properties**(struct *drm_device* *dev)
create TV connector margin properties

Parameters

struct drm_device *dev
DRM device

Description

Called by a driver's HDMI connector initialization routine, this function creates the TV margin properties for a given device. No need to call this function for an SDTV connector, it's already called from *drm_mode_create_tv_properties_legacy()*.

Return

0 on success or a negative error code on failure.

int **drm_mode_create_tv_properties_legacy**(struct *drm_device* *dev, unsigned int num_modes, const char *const modes[])
create TV specific connector properties

Parameters

struct drm_device *dev
DRM device

unsigned int num_modes
number of different TV formats (modes) supported

const char * const modes[]
array of pointers to strings containing name of each format

Description

Called by a driver's TV initialization routine, this function creates the TV specific connector properties for a given device. Caller is responsible for allocating a list of format names and passing them to this routine.

NOTE

This functions registers the deprecated “mode” connector property to select the analog TV mode (ie, NTSC, PAL, etc.). New drivers must use [`drm_mode_create_tv_properties\(\)`](#) instead.

Return

0 on success or a negative error code on failure.

int **drm_mode_create_tv_properties**(struct [`drm_device`](#) *dev, unsigned int supported_tv_modes)
create TV specific connector properties

Parameters

struct `drm_device` *dev
DRM device

unsigned int supported_tv_modes
Bitmask of TV modes supported (See `DRM_MODE_TV_MODE_*`)

Description

Called by a driver’s TV initialization routine, this function creates the TV specific connector properties for a given device.

Return

0 on success or a negative error code on failure.

int **drm_mode_create_scaling_mode_property**(struct [`drm_device`](#) *dev)
create scaling mode property

Parameters

struct `drm_device` *dev
DRM device

Description

Called by a driver the first time it’s needed, must be attached to desired connectors.

Atomic drivers should use [`drm_connector_attach_scaling_mode_property\(\)`](#) instead to correctly assign [`drm_connector_state.scaling_mode`](#) in the atomic state.

Return

0

int **drm_connector_attach_vrr_capable_property**(struct [`drm_connector`](#) *connector)
creates the vrr_capable property

Parameters

struct `drm_connector` *connector
connector to create the vrr_capable property on.

Description

This is used by atomic drivers to add support for querying variable refresh rate capability for a connector.

Return

Zero on success, negative errno on failure.

int **drm_connector_attach_scaling_mode_property**(struct *drm_connector* *connector, u32 scaling_mode_mask)

attach atomic scaling mode property

Parameters

struct drm_connector *connector
connector to attach scaling mode property on.

u32 scaling_mode_mask
or'ed mask of BIT(DRM_MODE_SCALE_*).

Description

This is used to add support for scaling mode to atomic drivers. The scaling mode will be set to *drm_connector_state.scaling_mode* and can be used from *drm_connector_helper_funcs->atomic_check* for validation.

This is the atomic version of *drm_mode_create_scaling_mode_property()*.

Return

Zero on success, negative errno on failure.

int **drm_mode_create_aspect_ratio_property**(struct *drm_device* *dev)
create aspect ratio property

Parameters

struct drm_device *dev
DRM device

Description

Called by a driver the first time it's needed, must be attached to desired connectors.

Return

Zero on success, negative errno on failure.

int **drm_mode_create_hdmi_colorspace_property**(struct *drm_connector* *connector, u32 supported_colorspaces)

create hdmi colorspace property

Parameters

struct drm_connector *connector
connector to create the Colorspace property on.

u32 supported_colorspaces
bitmap of supported color spaces

Description

Called by a driver the first time it's needed, must be attached to desired HDMI connectors.

Return

Zero on success, negative errno on failure.

int **drm_mode_create_dp_colorspace_property**(struct *drm_connector* *connector, u32 supported_colorspaces)

create dp colorspace property

Parameters

struct drm_connector *connector
connector to create the Colorspace property on.

u32 supported_colorspaces
bitmap of supported color spaces

Description

Called by a driver the first time it's needed, must be attached to desired DP connectors.

Return

Zero on success, negative errno on failure.

int **drm_mode_create_content_type_property**(struct *drm_device* *dev)
create content type property

Parameters

struct drm_device *dev
DRM device

Description

Called by a driver the first time it's needed, must be attached to desired connectors.

Return

Zero on success, negative errno on failure.

int **drm_mode_create_suggested_offset_properties**(struct *drm_device* *dev)
create suggests offset properties

Parameters

struct drm_device *dev
DRM device

Description

Create the suggested x/y offset property for connectors.

Return

0 on success or a negative error code on failure.

int **drm_connector_set_path_property**(struct *drm_connector* *connector, const char *path)
set tile property on connector

Parameters

struct drm_connector *connector
connector to set property on.

const char *path
path to use for property; must not be NULL.

Description

This creates a property to expose to userspace to specify a connector path. This is mainly used for DisplayPort MST where connectors have a topology and we want to allow userspace to give them more meaningful names.

Return

Zero on success, negative errno on failure.

int **drm_connector_set_tile_property**(struct *drm_connector* *connector)
set tile property on connector

Parameters

struct drm_connector *connector
connector to set property on.

Description

This looks up the tile information for a connector, and creates a property for userspace to parse if it exists. The property is of the form of 8 integers using ':' as a separator. This is used for dual port tiled displays with DisplayPort SST or DisplayPort MST connectors.

Return

Zero on success, errno on failure.

void **drm_connector_set_link_status_property**(struct *drm_connector* *connector, uint64_t link_status)
Set link status property of a connector

Parameters

struct drm_connector *connector
drm connector

uint64_t link_status
new value of link status property (0: Good, 1: Bad)

Description

In usual working scenario, this link status property will always be set to "GOOD". If something fails during or after a mode set, the kernel driver may set this link status property to "BAD". The caller then needs to send a hotplug uevent for userspace to re-check the valid modes through GET_CONNECTOR_IOCTL and retry modeset.

The reason for adding this property is to handle link training failures, but it is not limited to DP or link training. For example, if we implement asynchronous setcrtc, this property can be used to report any failures in that.

Note

Drivers cannot rely on userspace to support this property and issue a modeset. As such, they may choose to handle issues (like re-training a link) without userspace's intervention.

int **drm_connector_attach_max_bpc_property**(struct *drm_connector* *connector, int min, int max)
attach "max bpc" property

Parameters

struct drm_connector *connector

connector to attach max bpc property on.

int min

The minimum bit depth supported by the connector.

int max

The maximum bit depth supported by the connector.

Description

This is used to add support for limiting the bit depth on a connector.

Return

Zero on success, negative errno on failure.

int **drm_connector_attach_hdr_output_metadata_property**(struct *drm_connector* *connector)

attach "HDR_OUTPUT_METADATA" property

Parameters

struct drm_connector *connector

connector to attach the property on.

Description

This is used to allow the userspace to send HDR Metadata to the driver.

Return

Zero on success, negative errno on failure.

int **drm_connector_attach_colorspace_property**(struct *drm_connector* *connector)

attach "Colorspace" property

Parameters

struct drm_connector *connector

connector to attach the property on.

Description

This is used to allow the userspace to signal the output colorspace to the driver.

Return

Zero on success, negative errno on failure.

bool **drm_connector_atomic_hdr_metadata_equal**(struct *drm_connector_state* *old_state,
struct *drm_connector_state* *new_state)

checks if the hdr metadata changed

Parameters

struct drm_connector_state *old_state

old connector state to compare

struct drm_connector_state *new_state

new connector state to compare

Description

This is used by HDR-enabled drivers to test whether the HDR metadata have changed between two different connector state (and thus probably requires a full blown mode change).

Return

True if the metadata are equal, False otherwise

void **drm_connector_set_vrr_capable_property**(struct *drm_connector* *connector, bool capable)

sets the variable refresh rate capable property for a connector

Parameters

struct drm_connector *connector
drm connector

bool capable
True if the connector is variable refresh rate capable

Description

Should be used by atomic drivers to update the indicated support for variable refresh rate over a connector.

int **drm_connector_set_panel_orientation**(struct *drm_connector* *connector, enum *drm_panel_orientation* panel_orientation)

sets the connector's panel_orientation

Parameters

struct drm_connector *connector
connector for which to set the panel-orientation property.

enum drm_panel_orientation panel_orientation
drm_panel_orientation value to set

Description

This function sets the connector's panel_orientation and attaches a "panel orientation" property to the connector.

Calling this function on a connector where the panel_orientation has already been set is a no-op (e.g. the orientation has been overridden with a kernel commandline option).

It is allowed to call this function with a panel_orientation of `DRM_MODE_PANEL_ORIENTATION_UNKNOWN`, in which case it is a no-op.

The function shouldn't be called in panel after drm is registered (i.e. *drm_dev_register()* is called in drm).

Return

Zero on success, negative errno on failure.

int **drm_connector_set_panel_orientation_with_quirk**(struct *drm_connector* *connector, enum *drm_panel_orientation* panel_orientation, int width, int height)

set the connector's panel_orientation after checking for quirks

Parameters

struct drm_connector *connector
connector for which to init the panel-orientation property.

enum drm_panel_orientation panel_orientation
drm_panel_orientation value to set

int width
width in pixels of the panel, used for panel quirk detection

int height
height in pixels of the panel, used for panel quirk detection

Description

Like [drm_connector_set_panel_orientation\(\)](#), but with a check for platform specific (e.g. DMI based) quirks overriding the passed in panel_orientation.

Return

Zero on success, negative errno on failure.

int **drm_connector_set_orientation_from_panel**(struct [drm_connector](#) *connector, struct [drm_panel](#) *panel)
set the connector's panel_orientation from panel's callback.

Parameters

struct drm_connector *connector
connector for which to init the panel-orientation property.

struct drm_panel *panel
panel that can provide orientation information.

Description

Drm drivers should call this function before [drm_dev_register\(\)](#). Orientation is obtained from panel's .get_orientation() callback.

Return

Zero on success, negative errno on failure.

void **drm_connector_create_privacy_screen_properties**(struct [drm_connector](#) *connector)
create the drm connector's privacy-screen properties.

Parameters

struct drm_connector *connector
connector for which to create the privacy-screen properties

Description

This function creates the "privacy-screen sw-state" and "privacy-screen hw-state" properties for the connector. They are not attached.

void **drm_connector_attach_privacy_screen_properties**(struct [drm_connector](#) *connector)
attach the drm connector's privacy-screen properties.

Parameters

struct `drm_connector` *connector
connector on which to attach the privacy-screen properties

Description

This function attaches the “privacy-screen sw-state” and “privacy-screen hw-state” properties to the connector. The initial state of both is set to “Disabled”.

void **drm_connector_attach_privacy_screen_provider**(struct *drm_connector* *connector,
struct *drm_privacy_screen* *priv)

attach a privacy-screen to the connector

Parameters

struct `drm_connector` *connector
connector to attach the privacy-screen to

struct `drm_privacy_screen` *priv
drm_privacy_screen to attach

Description

Create and attach the standard privacy-screen properties and register a generic notifier for generating sysfs-connector-status-events on external changes to the privacy-screen status. This function takes ownership of the passed in `drm_privacy_screen` and will call *drm_privacy_screen_put()* on it when the connector is destroyed.

void **drm_connector_update_privacy_screen**(const struct *drm_connector_state*
*connector_state)

update connector’s privacy-screen sw-state

Parameters

const struct `drm_connector_state` *connector_state
connector-state to update the privacy-screen for

Description

This function calls *drm_privacy_screen_set_sw_state()* on the connector’s privacy-screen.

If the connector has no privacy-screen, then this is a no-op.

void **drm_connector_oob_hotplug_event**(struct *fwnode_handle* *connector_fwnode)

Report out-of-band hotplug event to connector

Parameters

struct `fwnode_handle` *connector_fwnode
fwnode_handle to report the event on

Description

On some hardware a hotplug event notification may come from outside the display driver / device. An example of this is some USB Type-C setups where the hardware muxes the DisplayPort data and aux-lines but does not pass the altmode HPD status bit to the GPU’s DP HPD pin.

This function can be used to report these out-of-band events after obtaining a `drm_connector` reference through calling *drm_connector_find_by_fwnode()*.

void **drm_mode_put_tile_group**(struct *drm_device* *dev, struct *drm_tile_group* *tg)
drop a reference to a tile group.

Parameters

struct drm_device *dev
DRM device

struct drm_tile_group *tg
tile group to drop reference to.

Description

drop reference to tile group and free if 0.

struct *drm_tile_group* ***drm_mode_get_tile_group**(struct *drm_device* *dev, const char topology[8])

get a reference to an existing tile group

Parameters

struct drm_device *dev
DRM device

const char topology[8]
8-bytes unique per monitor.

Description

Use the unique bytes to get a reference to an existing tile group.

Return

tile group or NULL if not found.

struct *drm_tile_group* ***drm_mode_create_tile_group**(struct *drm_device* *dev, const char topology[8])

create a tile group from a displayid description

Parameters

struct drm_device *dev
DRM device

const char topology[8]
8-bytes unique per monitor.

Description

Create a tile group for the unique monitor, and get a unique identifier for the tile group.

Return

new tile group or NULL.

4.11.2 Writeback Connectors

Writeback connectors are used to expose hardware which can write the output from a CRTC to a memory buffer. They are used and act similarly to other types of connectors, with some important differences:

- Writeback connectors don't provide a way to output visually to the user.
- Writeback connectors are visible to userspace only when the client sets `DRM_CLIENT_CAP_WRITEBACK_CONNECTORS`.
- Writeback connectors don't have EDID.

A framebuffer may only be attached to a writeback connector when the connector is attached to a CRTC. The `WRITEBACK_FB_ID` property which sets the framebuffer applies only to a single commit (see below). A framebuffer may not be attached while the CRTC is off.

Unlike with planes, when a writeback framebuffer is removed by userspace DRM makes no attempt to remove it from active use by the connector. This is because no method is provided to abort a writeback operation, and in any case making a new commit whilst a writeback is ongoing is undefined (see `WRITEBACK_OUT_FENCE_PTR` below). As soon as the current writeback is finished, the framebuffer will automatically no longer be in active use. As it will also have already been removed from the framebuffer list, there will be no way for any userspace application to retrieve a reference to it in the intervening period.

Writeback connectors have some additional properties, which userspace can use to query and control them:

“`WRITEBACK_FB_ID`”:

Write-only object property storing a `DRM_MODE_OBJECT_FB`: it stores the framebuffer to be written by the writeback connector. This property is similar to the `FB_ID` property on planes, but will always read as zero and is not preserved across commits. Userspace must set this property to an output buffer every time it wishes the buffer to get filled.

“`WRITEBACK_PIXEL_FORMATS`”:

Immutable blob property to store the supported pixel formats table. The data is an array of `u32 DRM_FORMAT_* fourcc` values. Userspace can use this blob to find out what pixel formats are supported by the connector's writeback engine.

“`WRITEBACK_OUT_FENCE_PTR`”:

Userspace can use this property to provide a pointer for the kernel to fill with a `sync_file` file descriptor, which will signal once the writeback is finished. The value should be the address of a 32-bit signed integer, cast to a `u64`. Userspace should wait for this fence to signal before making another commit affecting any of the same CRTCs, Planes or Connectors. **Failure to do so will result in undefined behaviour.** For this reason it is strongly recommended that all userspace applications making use of writeback connectors *always* retrieve an out-fence for the commit and use it appropriately. From userspace, this property will always read as zero.

```
struct drm_writeback_connector
```

DRM writeback connector

Definition:

```
struct drm_writeback_connector {
    struct drm_connector base;
    struct drm_encoder encoder;
    struct drm_property_blob *pixel_formats_blob_ptr;
    spinlock_t job_lock;
    struct list_head job_queue;
    unsigned int fence_context;
    spinlock_t fence_lock;
    unsigned long fence_seqno;
    char timeline_name[32];
};
```

Members

base

base `drm_connector` object

encoder

Internal encoder used by the connector to fulfill the DRM framework requirements. The users of the **drm_writeback_connector** control the behaviour of the **encoder** by passing the **enc_funcs** parameter to `drm_writeback_connector_init()` function. For users of `drm_writeback_connector_init_with_encoder()`, this field is not valid as the encoder is managed within their drivers.

pixel_formats_blob_ptr

DRM blob property data for the pixel formats list on writeback connectors See also `drm_writeback_connector_init()`

job_lock

Protects `job_queue`

job_queue

Holds a list of a connector's writeback jobs; the last item is the most recent. The first item may be either waiting for the hardware to begin writing, or currently being written.

See also: `drm_writeback_queue_job()` and `drm_writeback_signal_completion()`

fence_context

timeline context used for fence operations.

fence_lock

spinlock to protect the fences in the `fence_context`.

fence_seqno

Seqno variable used as monotonic counter for the fences created on the connector's timeline.

timeline_name

The name of the connector's fence timeline.

struct `drm_writeback_job`

DRM writeback job

Definition:

```
struct drm_writeback_job {
    struct drm_writeback_connector *connector;
```

```

bool prepared;
struct work_struct cleanup_work;
struct list_head list_entry;
struct drm_framebuffer *fb;
struct dma_fence *out_fence;
void *priv;
};

```

Members

connector

Back-pointer to the writeback connector associated with the job

prepared

Set when the job has been prepared with `drm_writeback_prepare_job()`

cleanup_work

Used to allow `drm_writeback_signal_completion` to defer dropping the framebuffer reference to a workqueue

list_entry

List item for the writeback connector's **job_queue**

fb

Framebuffer to be written to by the writeback connector. Do not set directly, use `drm_writeback_set_fb()`

out_fence

Fence which will signal once the writeback has completed

priv

Driver-private data

```

int drm_writeback_connector_init(struct drm_device *dev, struct drm_writeback_connector
                                *wb_connector, const struct drm_connector_funcs
                                *con_funcs, const struct drm_encoder_helper_funcs
                                *enc_helper_funcs, const u32 *formats, int n_formats,
                                u32 possible_crtcs)

```

Initialize a writeback connector and its properties

Parameters

struct *drm_device* *dev

DRM device

struct *drm_writeback_connector* *wb_connector

Writeback connector to initialize

const struct *drm_connector_funcs* *con_funcs

Connector funcs vtable

const struct *drm_encoder_helper_funcs* *enc_helper_funcs

Encoder helper funcs vtable to be used by the internal encoder

const u32 *formats

Array of supported pixel formats for the writeback engine

int n_formats

Length of the formats array

u32 possible_crtcs

possible crtcs for the internal writeback encoder

Description

This function creates the writeback-connector-specific properties if they have not been already created, initializes the connector as type `DRM_MODE_CONNECTOR_WRITEBACK`, and correctly initializes the property values. It will also create an internal encoder associated with the `drm_writeback_connector` and set it to use the **enc_helper_funcs** vtable for the encoder helper.

Drivers should always use this function instead of `drm_connector_init()` to set up writeback connectors.

Return

0 on success, or a negative error code

```
int drm_writeback_connector_init_with_encoder(struct drm_device *dev, struct
                                             drm_writeback_connector
                                             *wb_connector, struct drm_encoder
                                             *enc, const struct drm_connector_funcs
                                             *con_funcs, const u32 *formats, int
                                             n_formats)
```

Initialize a writeback connector with a custom encoder

Parameters

struct *drm_device* *dev

DRM device

struct *drm_writeback_connector* *wb_connector

Writeback connector to initialize

struct *drm_encoder* *enc

handle to the already initialized drm encoder

const struct *drm_connector_funcs* *con_funcs

Connector funcs vtable

const u32 *formats

Array of supported pixel formats for the writeback engine

int n_formats

Length of the formats array

Description

This function creates the writeback-connector-specific properties if they have not been already created, initializes the connector as type `DRM_MODE_CONNECTOR_WRITEBACK`, and correctly initializes the property values.

This function assumes that the `drm_writeback_connector`'s encoder has already been created and initialized before invoking this function.

In addition, this function also assumes that callers of this API will manage assigning the encoder helper functions, `possible_crtcs` and any other encoder specific operation.

Drivers should always use this function instead of `drm_connector_init()` to set up writeback connectors if they want to manage themselves the lifetime of the associated encoder.

Return

0 on success, or a negative error code

```
void drm_writeback_queue_job(struct drm_writeback_connector *wb_connector, struct  
                             drm_connector_state *conn_state)
```

Queue a writeback job for later signalling

Parameters

```
struct drm_writeback_connector *wb_connector
```

The writeback connector to queue a job on

```
struct drm_connector_state *conn_state
```

The connector state containing the job to queue

Description

This function adds the job contained in **conn_state** to the job_queue for a writeback connector. It takes ownership of the writeback job and sets the **conn_state->writeback_job** to NULL, and so no access to the job may be performed by the caller after this function returns.

Drivers must ensure that for a given writeback connector, jobs are queued in exactly the same order as they will be completed by the hardware (and signaled via `drm_writeback_signal_completion()`).

For every call to `drm_writeback_queue_job()` there must be exactly one call to `drm_writeback_signal_completion()`

See also: `drm_writeback_signal_completion()`

```
void drm_writeback_signal_completion(struct drm_writeback_connector *wb_connector,  
                                     int status)
```

Signal the completion of a writeback job

Parameters

```
struct drm_writeback_connector *wb_connector
```

The writeback connector whose job is complete

```
int status
```

Status code to set in the writeback out_fence (0 for success)

Description

Drivers should call this to signal the completion of a previously queued writeback job. It should be called as soon as possible after the hardware has finished writing, and may be called from interrupt context. It is the driver's responsibility to ensure that for a given connector, the hardware completes writeback jobs in the same order as they are queued.

Unless the driver is holding its own reference to the framebuffer, it must not be accessed after calling this function.

See also: `drm_writeback_queue_job()`

4.12 Encoder Abstraction

Encoders represent the connecting element between the CRTC (as the overall pixel pipeline, represented by `struct drm_crtc`) and the connectors (as the generic sink entity, represented by `struct drm_connector`). An encoder takes pixel data from a CRTC and converts it to a format suitable for any attached connector. Encoders are objects exposed to userspace, originally to allow userspace to infer cloning and connector/CRTC restrictions. Unfortunately almost all drivers get this wrong, making the uabi pretty much useless. On top of that the exposed restrictions are too simple for today's hardware, and the recommended way to infer restrictions is by using the `DRM_MODE_ATOMIC_TEST_ONLY` flag for the atomic IOCTL.

Otherwise encoders aren't used in the uapi at all (any modeset request from userspace directly connects a connector with a CRTC), drivers are therefore free to use them however they wish. Modeset helper libraries make strong use of encoders to facilitate code sharing. But for more complex settings it is usually better to move shared code into a separate `drm_bridge`. Compared to encoders, bridges also have the benefit of being purely an internal abstraction since they are not exposed to userspace at all.

Encoders are initialized with `drm_encoder_init()` and cleaned up using `drm_encoder_cleanup()`.

4.12.1 Encoder Functions Reference

struct **drm_encoder_funcs**
encoder controls

Definition:

```
struct drm_encoder_funcs {
    void (*reset)(struct drm_encoder *encoder);
    void (*destroy)(struct drm_encoder *encoder);
    int (*late_register)(struct drm_encoder *encoder);
    void (*early_unregister)(struct drm_encoder *encoder);
};
```

Members

reset

Reset encoder hardware and software state to off. This function isn't called by the core directly, only through `drm_mode_config_reset()`. It's not a helper hook only for historical reasons.

destroy

Clean up encoder resources. This is only called at driver unload time through `drm_mode_config_cleanup()` since an encoder cannot be hotplugged in DRM.

late_register

This optional hook can be used to register additional userspace interfaces attached to the encoder like debugfs interfaces. It is called late in the driver load sequence from `drm_dev_register()`. Everything added from this callback should be unregistered in the `early_unregister` callback.

Returns:

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

early_unregister

This optional hook should be used to unregister the additional userspace interfaces attached to the encoder from **late_register**. It is called from `drm_dev_unregister()`, early in the driver unload sequence to disable userspace access before data structures are torn-down.

Description

Encoders sit between CRTC's and connectors.

struct drm_encoder

central DRM encoder structure

Definition:

```
struct drm_encoder {
    struct drm_device *dev;
    struct list_head head;
    struct drm_mode_object base;
    char *name;
    int encoder_type;
    unsigned index;
    uint32_t possible_crtcs;
    uint32_t possible_clones;
    struct drm_crtc *crtc;
    struct list_head bridge_chain;
    const struct drm_encoder_funcs *funcs;
    const struct drm_encoder_helper_funcs *helper_private;
};
```

Members

dev

parent DRM device

head

list management

base

base KMS object

name

human readable name, can be overwritten by the driver

encoder_type

One of the `DRM_MODE_ENCODER_<foo>` types in `drm_mode.h`. The following encoder types are defined thus far:

- `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, or in general any panel with a proprietary parallel connector.

- `DRM_MODE_ENCODER_TVDAC` for TV output (Composite, S-Video, Component, SCART).
- `DRM_MODE_ENCODER_VIRTUAL` for virtual machine displays
- `DRM_MODE_ENCODER_DSI` for panels connected using the DSI serial bus.
- `DRM_MODE_ENCODER_DPI` for panels connected using the DPI parallel bus.
- `DRM_MODE_ENCODER_DPMST` for special fake encoders used to allow multiple DP MST streams to share one physical encoder.

index

Position inside the `mode_config.list`, can be used as an array index. It is invariant over the lifetime of the encoder.

possible_crtcs

Bitmask of potential CRTC bindings, using `drm_crtc_index()` as the index into the bitfield. The driver must set the bits for all `drm_crtc` objects this encoder can be connected to before calling `drm_dev_register()`.

You will get a WARN if you get this wrong in the driver.

Note that since CRTC objects can't be hotplugged the assigned indices are stable and hence known before registering all objects.

possible_clones

Bitmask of potential sibling encoders for cloning, using `drm_encoder_index()` as the index into the bitfield. The driver must set the bits for all `drm_encoder` objects which can clone a `drm_crtc` together with this encoder before calling `drm_dev_register()`. Drivers should set the bit representing the encoder itself, too. Cloning bits should be set such that when two encoders can be used in a cloned configuration, they both should have each other bits set.

As an exception to the above rule if the driver doesn't implement any cloning it can leave **possible_clones** set to 0. The core will automatically fix this up by setting the bit for the encoder itself.

You will get a WARN if you get this wrong in the driver.

Note that since encoder objects can't be hotplugged the assigned indices are stable and hence known before registering all objects.

crtc

Currently bound CRTC, only really meaningful for non-atomic drivers. Atomic drivers should instead check `drm_connector_state.crtc`.

bridge_chain

Bridges attached to this encoder. Drivers shall not access this field directly.

funcs

control functions, can be NULL for simple managed encoders

helper_private

mid-layer private data

Description

CRTCs drive pixels to encoders, which convert them into signals appropriate for a given connector or set of connectors.

drm_encoder_alloc

`drm_encoder_alloc (dev, type, member, funcs, encoder_type, name, ...)`

Allocate and initialize an encoder

Parameters**dev**

drm device

type

the type of the struct which contains struct *drm_encoder*

member

the name of the *drm_encoder* within **type**

funcs

callbacks for this encoder (optional)

encoder_type

user visible type of the encoder

name

printf style format string for the encoder name, or NULL for default name

...

variable arguments

Description

Allocates and initializes an encoder. Encoder should be subclassed as part of driver encoder objects. Cleanup is automatically handled through registering *drm_encoder_cleanup()* with *drm_add_action()*.

The **drm_encoder_funcs.destroy** hook must be NULL.

Return

Pointer to new encoder, or ERR_PTR on failure.

drm_plain_encoder_alloc

`drm_plain_encoder_alloc (dev, funcs, encoder_type, name, ...)`

Allocate and initialize an encoder

Parameters**dev**

drm device

funcs

callbacks for this encoder (optional)

encoder_type

user visible type of the encoder

name

printf style format string for the encoder name, or NULL for default name

...

variable arguments

Description

This is a simplified version of `drm_encoder_alloc()`, which only allocates and returns a `struct drm_encoder` instance, with no subclassing.

Return

Pointer to the new `drm_encoder` struct, or `ERR_PTR` on failure.

unsigned int **drm_encoder_index**(const struct `drm_encoder` *encoder)
find the index of a registered encoder

Parameters

const struct drm_encoder *encoder
encoder to find index for

Description

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

u32 **drm_encoder_mask**(const struct `drm_encoder` *encoder)
find the mask of a registered encoder

Parameters

const struct drm_encoder *encoder
encoder to find mask for

Description

Given a registered encoder, return the mask bit of that encoder for an encoder's possible_clones field.

bool **drm_encoder_crtc_ok**(struct `drm_encoder` *encoder, struct `drm_crtc` *crtc)
can a given crtc drive a given encoder?

Parameters

struct drm_encoder *encoder
encoder to test

struct drm_crtc *crtc
crtc to test

Description

Returns false if **encoder** can't be driven by **crtc**, true otherwise.

struct `drm_encoder` ***drm_encoder_find**(struct `drm_device` *dev, struct `drm_file` *file_priv, uint32_t id)
find a `drm_encoder`

Parameters

struct drm_device *dev
DRM device

struct drm_file *file_priv
drm file to check for lease against.

uint32_t id
encoder id

Description

Returns the encoder with **id**, NULL if it doesn't exist. Simple wrapper around [drm_mode_object_find\(\)](#).

drm_for_each_encoder_mask

drm_for_each_encoder_mask (encoder, dev, encoder_mask)
iterate over encoders specified by bitmask

Parameters

encoder
the loop cursor

dev
the DRM device

encoder_mask
bitmask of encoder indices

Description

Iterate over all encoders specified by bitmask.

drm_for_each_encoder

drm_for_each_encoder (encoder, dev)
iterate over all encoders

Parameters

encoder
the loop cursor

dev
the DRM device

Description

Iterate over all encoders of **dev**.

int **drm_encoder_init**(struct [drm_device](#) *dev, struct [drm_encoder](#) *encoder, const struct [drm_encoder_funcs](#) *funcs, int encoder_type, const char *name, ...)
Init a preallocated encoder

Parameters

struct drm_device *dev
drm device

struct drm_encoder *encoder
the encoder to init

const struct drm_encoder_funcs *funcs
callbacks for this encoder

int encoder_type

user visible type of the encoder

const char *name

printf style format string for the encoder name, or NULL for default name

...

variable arguments

Description

Initializes a preallocated encoder. Encoder should be subclassed as part of driver encoder objects. At driver unload time the driver's *drm_encoder_funcs.destroy* hook should call *drm_encoder_cleanup()* and *kfree()* the encoder structure. The encoder structure should not be allocated with *devm_kzalloc()*.

Note

consider using *drm_encoder_alloc()* or *drm_encoder_init()* instead of *drm_encoder_init()* to let the DRM managed resource infrastructure take care of cleanup and deallocation.

Return

Zero on success, error code on failure.

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

cleans up an initialised encoder

Parameters

struct drm_encoder *encoder

encoder to cleanup

Description

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

int **drm_encoder_init**(struct *drm_device* *dev, struct *drm_encoder* *encoder, const struct *drm_encoder_funcs* *funcs, int encoder_type, const char *name, ...)

Initialize a preallocated encoder

Parameters

struct drm_device *dev

drm device

struct drm_encoder *encoder

the encoder to init

const struct drm_encoder_funcs *funcs

callbacks for this encoder (optional)

int encoder_type

user visible type of the encoder

const char *name

printf style format string for the encoder name, or NULL for default name

...

variable arguments

Description

Initializes a preallocated encoder. Encoder should be subclassed as part of driver encoder objects. Cleanup is automatically handled through registering `drm_encoder_cleanup()` with `drm_add_action()`. The encoder structure should be allocated with `drm_kzalloc()`.

The `drm_encoder_funcs.destroy` hook must be NULL.

Return

Zero on success, error code on failure.

4.13 KMS Locking

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

For basic principles of `ww_mutex`, see: Documentation/locking/ww-mutex-design.rst

The basic usage pattern is to:

```
drm_modeset_acquire_init(ctx, DRM_MODESET_ACQUIRE_INTERRUPTIBLE)
retry:
foreach (lock in random_ordered_set_of_locks) {
    ret = drm_modeset_lock(lock, ctx)
    if (ret == -EDEADLK) {
        ret = drm_modeset_backoff(ctx);
        if (!ret)
            goto retry;
    }
    if (ret)
        goto out;
}
... do stuff ...
out:
drm_modeset_drop_locks(ctx);
drm_modeset_acquire_fini(ctx);
```

For convenience this control flow is implemented in `DRM_MODESET_LOCK_ALL_BEGIN()` and `DRM_MODESET_LOCK_ALL_END()` for the case where all modeset locks need to be taken through `drm_modeset_lock_all_ctx()`.

If all that is needed is a single modeset lock, then the `struct drm_modeset_acquire_ctx` is not needed and the locking can be simplified by passing a NULL instead of `ctx` in the `drm_modeset_lock()` call or calling `drm_modeset_lock_single_interruptible()`. To unlock afterwards call `drm_modeset_unlock()`.

On top of these per-object locks using `ww_mutex` there's also an overall `drm_mode_config.mutex`, for protecting everything else. Mostly this means probe state of connectors, and preventing hotplug add/removal of connectors.

Finally there's a bunch of dedicated locks to protect drm core internal lists and lookup data structures.

struct **drm_modeset_acquire_ctx**
locking context (see `ww_acquire_ctx`)

Definition:

```
struct drm_modeset_acquire_ctx {
    struct ww_acquire_ctx ww_ctx;
    struct drm_modeset_lock *contended;
    depot_stack_handle_t stack_depot;
    struct list_head locked;
    bool trylock_only;
    bool interruptible;
};
```

Members

ww_ctx
base acquire ctx

contended
used internally for -EDEADLK handling

stack_depot
used internally for contention debugging

locked
list of held locks

trylock_only
trylock mode used in atomic contexts/panic notifiers

interruptible
whether interruptible locking should be used.

Description

Each thread competing for a set of locks must use one acquire ctx. And if any lock fxn returns -EDEADLK, it must backoff and retry.

struct **drm_modeset_lock**
used for locking modeset resources.

Definition:

```
struct drm_modeset_lock {
    struct ww_mutex mutex;
    struct list_head head;
};
```

Members

mutex
resource locking

head

used to hold its place on `drm_atomi_state.locked` list when part of an atomic update

Description

Used for locking CRTC's and other modeset resources.

void **drm_modeset_lock_fini**(struct *drm_modeset_lock* *lock)
cleanup lock

Parameters

struct drm_modeset_lock *lock
lock to cleanup

bool **drm_modeset_is_locked**(struct *drm_modeset_lock* *lock)
equivalent to `mutex_is_locked()`

Parameters

struct drm_modeset_lock *lock
lock to check

void **drm_modeset_lock_assert_held**(struct *drm_modeset_lock* *lock)
equivalent to `lockdep_assert_held()`

Parameters

struct drm_modeset_lock *lock
lock to check

DRM_MODESET_LOCK_ALL_BEGIN

DRM_MODESET_LOCK_ALL_BEGIN (dev, ctx, flags, ret)
Helper to acquire modeset locks

Parameters

dev
drm device

ctx
local modeset acquire context, will be dereferenced

flags
DRM_MODESET_ACQUIRE_* flags to pass to *drm_modeset_acquire_init()*

ret
local ret/err/etc variable to track error status

Description

Use these macros to simplify grabbing all modeset locks using a local context. This has the advantage of reducing boilerplate, but also properly checking return values where appropriate.

Any code run between BEGIN and END will be holding the modeset locks.

This must be paired with *DRM_MODESET_LOCK_ALL_END()*. We will jump back and forth between the labels on deadlock and error conditions.

Drivers can acquire additional modeset locks. If any lock acquisition fails, the control flow needs to jump to `DRM_MODESET_LOCK_ALL_END()` with the `ret` parameter containing the return value of `drm_modeset_lock()`.

Return

The only possible value of `ret` immediately after `DRM_MODESET_LOCK_ALL_BEGIN()` is 0, so no error checking is necessary

DRM_MODESET_LOCK_ALL_END

`DRM_MODESET_LOCK_ALL_END (dev, ctx, ret)`

Helper to release and cleanup modeset locks

Parameters

`dev`

drm device

`ctx`

local modeset acquire context, will be dereferenced

`ret`

local ret/err/etc variable to track error status

Description

The other side of `DRM_MODESET_LOCK_ALL_BEGIN()`. It will bounce back to BEGIN if `ret` is -EDEADLK.

It's important that you use the same `ret` variable for begin and end so deadlock conditions are properly handled.

Return

`ret` will be untouched unless it is -EDEADLK on entry. That means that if you successfully acquire the locks, `ret` will be whatever your code sets it to. If there is a deadlock or other failure with acquire or backoff, `ret` will be set to that failure. In both of these cases the code between BEGIN/END will not be run, so the failure will reflect the inability to grab the locks.

void `drm_modeset_lock_all`(struct `drm_device` *dev)

take all modeset locks

Parameters

`struct drm_device` *dev

DRM device

Description

This function takes all modeset locks, suitable where a more fine-grained scheme isn't (yet) implemented. Locks must be dropped by calling the `drm_modeset_unlock_all()` function.

This function is deprecated. It allocates a lock acquisition context and stores it in `drm_device.mode_config`. This facilitate conversion of existing code because it removes the need to manually deal with the acquisition context, but it is also brittle because the context is global and care must be taken not to nest calls. New code should use the `drm_modeset_lock_all_ctx()` function and pass in the context explicitly.

void **drm_modeset_unlock_all**(struct *drm_device* *dev)
drop all modeset locks

Parameters

struct drm_device *dev
DRM device

Description

This function drops all modeset locks taken by a previous call to the *drm_modeset_lock_all()* function.

This function is deprecated. It uses the lock acquisition context stored in *drm_device.mode_config*. This facilitates conversion of existing code because it removes the need to manually deal with the acquisition context, but it is also brittle because the context is global and care must be taken not to nest calls. New code should pass the acquisition context directly to the *drm_modeset_drop_locks()* function.

void **drm_warn_on_modeset_not_all_locked**(struct *drm_device* *dev)
check that all modeset locks are locked

Parameters

struct drm_device *dev
device

Description

Useful as a debug assert.

void **drm_modeset_acquire_init**(struct *drm_modeset_acquire_ctx* *ctx, uint32_t flags)
initialize acquire context

Parameters

struct drm_modeset_acquire_ctx *ctx
the acquire context

uint32_t flags
0 or `DRM_MODESET_ACQUIRE_INTERRUPTIBLE`

Description

When passing `DRM_MODESET_ACQUIRE_INTERRUPTIBLE` to **flags**, all calls to *drm_modeset_lock()* will perform an interruptible wait.

void **drm_modeset_acquire_fini**(struct *drm_modeset_acquire_ctx* *ctx)
cleanup acquire context

Parameters

struct drm_modeset_acquire_ctx *ctx
the acquire context

void **drm_modeset_drop_locks**(struct *drm_modeset_acquire_ctx* *ctx)
drop all locks

Parameters

struct drm_modeset_acquire_ctx *ctx
the acquire context

Description

Drop all locks currently held against this acquire context.

int **drm_modeset_backoff**(struct *drm_modeset_acquire_ctx* *ctx)
deadlock avoidance backoff

Parameters

struct drm_modeset_acquire_ctx *ctx
the acquire context

Description

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

This function returns 0 on success, or -ERESTARTSYS if this context is initialized with DRM_MODESET_ACQUIRE_INTERRUPTIBLE and the wait has been interrupted.

void **drm_modeset_lock_init**(struct *drm_modeset_lock* *lock)
initialize lock

Parameters

struct drm_modeset_lock *lock
lock to init

int **drm_modeset_lock**(struct *drm_modeset_lock* *lock, struct *drm_modeset_acquire_ctx* *ctx)
take modeset lock

Parameters

struct drm_modeset_lock *lock
lock to take

struct drm_modeset_acquire_ctx *ctx
acquire ctx

Description

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

If the **ctx** is not NULL and initialized with DRM_MODESET_ACQUIRE_INTERRUPTIBLE, this function will fail with -ERESTARTSYS when interrupted.

If **ctx** is NULL then the function call behaves like a normal, uninterruptible non-nesting mutex_lock() call.

int **drm_modeset_lock_single_interruptible**(struct *drm_modeset_lock* *lock)
take a single modeset lock

Parameters

struct drm_modeset_lock *lock
lock to take

Description

This function behaves as [`drm_modeset_lock\(\)`](#) with a NULL context, but performs interruptible waits.

This function returns 0 on success, or -ERESTARTSYS when interrupted.

```
void drm_modeset_unlock(struct drm\_modeset\_lock *lock)
    drop modeset lock
```

Parameters

```
struct drm_modeset_lock *lock
    lock to release
```

```
int drm_modeset_lock_all_ctx(struct drm\_device *dev, struct drm\_modeset\_acquire\_ctx
                             *ctx)
    take all modeset locks
```

Parameters

```
struct drm_device *dev
    DRM device
```

```
struct drm_modeset_acquire_ctx *ctx
    lock acquisition context
```

Description

This function takes all modeset locks, suitable where a more fine-grained scheme isn't (yet) implemented.

Unlike [`drm_modeset_lock_all\(\)`](#), it doesn't take the [`drm_mode_config.mutex`](#) since that lock isn't required for modeset state changes. Callers which need to grab that lock too need to do so outside of the acquire context **ctx**.

Locks acquired with this function should be released by calling the [`drm_modeset_drop_locks\(\)`](#) function on **ctx**.

See also: [`DRM_MODESET_LOCK_ALL_BEGIN\(\)`](#) and [`DRM_MODESET_LOCK_ALL_END\(\)`](#)

Return

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

4.14 KMS Properties

This section of the documentation is primarily aimed at user-space developers. For the driver APIs, see the other sections.

4.14.1 Requirements

KMS drivers might need to add extra properties to support new features. Each new property introduced in a driver needs to meet a few requirements, in addition to the one mentioned above:

- It must be standardized, documenting:
 - The full, exact, name string;
 - If the property is an enum, all the valid value name strings;
 - What values are accepted, and what these values mean;
 - What the property does and how it can be used;
 - How the property might interact with other, existing properties.
- It must provide a generic helper in the core code to register that property on the object it attaches to.
- Its content must be decoded by the core and provided in the object's associated state structure. That includes anything drivers might want to precompute, like `struct drm_clip_rect` for planes.
- Its initial state must match the behavior prior to the property introduction. This might be a fixed value matching what the hardware does, or it may be inherited from the state the firmware left the system in during boot.
- An IGT test must be submitted where reasonable.

4.14.2 Property Types and Blob Property Support

Properties as represented by `drm_property` are used to extend the modeset interface exposed to userspace. For the atomic modeset IOCTL properties are even the only way to transport metadata about the desired new modeset configuration from userspace to the kernel. Properties have a well-defined value range, which is enforced by the drm core. See the documentation of the flags member of `struct drm_property` for an overview of the different property types and ranges.

Properties don't store the current value directly, but need to be instantiated by attaching them to a `drm_mode_object` with `drm_object_attach_property()`.

Property values are only 64bit. To support bigger piles of data (like gamma tables, color correction matrices or large structures) a property can instead point at a `drm_property_blob` with that additional data.

Properties are defined by their symbolic name, userspace must keep a per-object mapping from those names to the property ID used in the atomic IOCTL and in the get/set property IOCTL.

`struct drm_property_enum`

symbolic values for enumerations

Definition:

```
struct drm_property_enum {
    uint64_t value;
    struct list_head head;
```

```
char name[DRM_PROP_NAME_LEN];
};
```

Members

value

numeric property value for this enum entry

If the property has the type `DRM_MODE_PROP_BITMASK`, **value** stores a bitshift, not a bitmask. In other words, the enum entry is enabled if the bit number **value** is set in the property's value. This enum entry has the bitmask `1 << value`.

head

list of enum values, linked to *`drm_property.enum_list`*

name

symbolic name for the enum

Description

For enumeration and bitmask properties this structure stores the symbolic decoding for each value. This is used for example for the rotation property.

struct `drm_property`

modeset object property

Definition:

```
struct drm_property {
    struct list_head head;
    struct drm_mode_object base;
    uint32_t flags;
    char name[DRM_PROP_NAME_LEN];
    uint32_t num_values;
    uint64_t *values;
    struct drm_device *dev;
    struct list_head enum_list;
};
```

Members

head

per-device list of properties, for cleanup.

base

base KMS object

flags

Property flags and type. A property needs to be one of the following types:

DRM_MODE_PROP_RANGE

Range properties report their minimum and maximum admissible unsigned values. The KMS core verifies that values set by application fit in that range. The range is unsigned. Range properties are created using *`drm_property_create_range()`*.

DRM_MODE_PROP_SIGNED_RANGE

Range properties report their minimum and maximum admissible unsigned values.

The KMS core verifies that values set by application fit in that range. The range is signed. Range properties are created using `drm_property_create_signed_range()`.

DRM_MODE_PROP_ENUM

Enumerated properties take a numerical value that ranges from 0 to the number of enumerated values defined by the property minus one, and associate a free-formed string name to each value. Applications can retrieve the list of defined value-name pairs and use the numerical value to get and set property instance values. Enum properties are created using `drm_property_create_enum()`.

DRM_MODE_PROP_BITMASK

Bitmask properties are enumeration properties that additionally restrict all enumerated values to the 0..63 range. Bitmask property instance values combine one or more of the enumerated bits defined by the property. Bitmask properties are created using `drm_property_create_bitmask()`.

DRM_MODE_PROP_OBJECT

Object properties are used to link modeset objects. This is used extensively in the atomic support to create the display pipeline, by linking `drm_framebuffer` to `drm_plane`, `drm_plane` to `drm_crtc` and `drm_connector` to `drm_crtc`. An object property can only link to a specific type of `drm_mode_object`, this limit is enforced by the core. Object properties are created using `drm_property_create_object()`.

Object properties work like blob properties, but in a more general fashion. They are limited to atomic drivers and must have the `DRM_MODE_PROP_ATOMIC` flag set.

DRM_MODE_PROP_BLOB

Blob properties store a binary blob without any format restriction. The binary blobs are created as KMS standalone objects, and blob property instance values store the ID of their associated blob object. Blob properties are created by calling `drm_property_create()` with `DRM_MODE_PROP_BLOB` as the type.

Actual blob objects to contain blob data are created using `drm_property_create_blob()`, or through the corresponding IOCTL.

Besides the built-in limit to only accept blob objects blob properties work exactly like object properties. The only reasons blob properties exist is backwards compatibility with existing userspace.

In addition a property can have any combination of the below flags:

DRM_MODE_PROP_ATOMIC

Set for properties which encode atomic modeset state. Such properties are not exposed to legacy userspace.

DRM_MODE_PROP_IMMUTABLE

Set for properties whose values cannot be changed by userspace. The kernel is allowed to update the value of these properties. This is generally used to expose probe state to userspace, e.g. the EDID, or the connector path property on DP MST sinks. Kernel can update the value of an immutable property by calling `drm_object_property_set_value()`.

name

symbolic name of the properties

num_values

size of the **values** array.

values

Array with limits and values for the property. The interpretation of these limits is dependent upon the type per **flags**.

dev

DRM device

enum_list

List of `drm_prop_enum_list` structures with the symbolic names for enum and bitmask values.

Description

This structure represent a modeset object property. It combines both the name of the property with the set of permissible values. This means that when a driver wants to use a property with the same name on different objects, but with different value ranges, then it must create property for each one. An example would be rotation of `drm_plane`, when e.g. the primary plane cannot be rotated. But if both the name and the value range match, then the same property structure can be instantiated multiple times for the same object. Userspace must be able to cope with this and cannot assume that the same symbolic property will have the same modeset object ID on all modeset objects.

Properties are created by one of the special functions, as explained in detail in the **flags** structure member.

To actually expose a property it must be attached to each object using `drm_object_attach_property()`. Currently properties can only be attached to `drm_connector`, `drm_crtc` and `drm_plane`.

Properties are also used as the generic metadata transport for the atomic IOCTL. Everything that was set directly in structures in the legacy modeset IOCTLs (like the plane source or destination windows, or e.g. the links to the CRTC) is exposed as a property with the `DRM_MODE_PROP_ATOMIC` flag set.

struct `drm_property_blob`

Blob data for `drm_property`

Definition:

```
struct drm_property_blob {
    struct drm_mode_object base;
    struct drm_device *dev;
    struct list_head head_global;
    struct list_head head_file;
    size_t length;
    void *data;
};
```

Members**base**

base KMS object

dev

DRM device

head_global

entry on the global blob list in *drm_mode_config.property_blob_list*.

head_file

entry on the per-file blob list in *drm_file.blobs* list.

length

size of the blob in bytes, invariant over the lifetime of the object

data

actual data, embedded at the end of this structure

Description

Blobs are used to store bigger values than what fits directly into the 64 bits available for a *drm_property*.

Blobs are reference counted using *drm_property_blob_get()* and *drm_property_blob_put()*. They are created using *drm_property_create_blob()*.

bool **drm_property_type_is**(struct *drm_property* *property, uint32_t type)

check the type of a property

Parameters

struct *drm_property* *property

property to check

uint32_t type

property type to compare with

Description

This is a helper function because the uapi encoding of property types is a bit special for historical reasons.

struct *drm_property* ***drm_property_find**(struct *drm_device* *dev, struct *drm_file* *file_priv, uint32_t id)

find property object

Parameters

struct *drm_device* *dev

DRM device

struct *drm_file* *file_priv

drm file to check for lease against.

uint32_t id

property object id

Description

This function looks up the property object specified by id and returns it.

struct *drm_property* ***drm_property_create**(struct *drm_device* *dev, u32 flags, const char *name, int num_values)

create a new property type

Parameters

struct drm_device *dev

drm device

u32 flags

flags specifying the property type

const char *name

name of the property

int num_values

number of pre-defined values

Description

This creates a new generic drm property which can then be attached to a drm object with [drm_object_attach_property\(\)](#). The returned property object must be freed with [drm_property_destroy\(\)](#), which is done automatically when calling [drm_mode_config_cleanup\(\)](#).

Return

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

struct [drm_property](#) ***drm_property_create_enum**(struct [drm_device](#) *dev, u32 flags, const char *name, const struct [drm_prop_enum_list](#) *props, int num_values)

create a new enumeration property type

Parameters

struct drm_device *dev

drm device

u32 flags

flags specifying the property type

const char *name

name of the property

const struct drm_prop_enum_list *props

enumeration lists with property values

int num_values

number of pre-defined values

Description

This creates a new generic drm property which can then be attached to a drm object with [drm_object_attach_property\(\)](#). The returned property object must be freed with [drm_property_destroy\(\)](#), which is done automatically when calling [drm_mode_config_cleanup\(\)](#).

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

Return

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

struct *drm_property* ***drm_property_create_bitmask**(struct *drm_device* *dev, u32 flags, const char *name, const struct drm_prop_enum_list *props, int num_props, uint64_t supported_bits)

create a new bitmask property type

Parameters

struct drm_device *dev
drm device

u32 flags
flags specifying the property type

const char *name
name of the property

const struct drm_prop_enum_list *props
enumeration lists with property bitflags

int num_props
size of the **props** array

uint64_t supported_bits
bitmask of all supported enumeration values

Description

This creates a new bitmask drm property which can then be attached to a drm object with *drm_object_attach_property()*. The returned property object must be freed with *drm_property_destroy()*, which is done automatically when calling *drm_mode_config_cleanup()*.

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

Return

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

struct *drm_property* ***drm_property_create_range**(struct *drm_device* *dev, u32 flags, const char *name, uint64_t min, uint64_t max)

create a new unsigned ranged property type

Parameters

struct drm_device *dev
drm device

u32 flags
flags specifying the property type

const char *name
name of the property

uint64_t min
minimum value of the property

uint64_t max
maximum value of the property

Description

This creates a new generic drm property which can then be attached to a drm object with `drm_object_attach_property()`. The returned property object must be freed with `drm_property_destroy()`, which is done automatically when calling `drm_mode_config_cleanup()`.

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

Return

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

```
struct drm_property *drm_property_create_signed_range(struct drm_device *dev, u32
                                                    flags, const char *name, int64_t
                                                    min, int64_t max)
```

create a new signed ranged property type

Parameters

struct *drm_device* *dev
drm device

u32 flags
flags specifying the property type

const char *name
name of the property

int64_t min
minimum value of the property

int64_t max
maximum value of the property

Description

This creates a new generic drm property which can then be attached to a drm object with `drm_object_attach_property()`. The returned property object must be freed with `drm_property_destroy()`, which is done automatically when calling `drm_mode_config_cleanup()`.

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

Return

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

```
struct drm_property *drm_property_create_object(struct drm_device *dev, u32 flags, const
                                                    char *name, uint32_t type)
```

create a new object property type

Parameters

struct *drm_device* *dev
drm device

u32 flags
flags specifying the property type

const char *name

name of the property

uint32_t type

object type from `DRM_MODE_OBJECT_*` defines

Description

This creates a new generic drm property which can then be attached to a drm object with `drm_object_attach_property()`. The returned property object must be freed with `drm_property_destroy()`, which is done automatically when calling `drm_mode_config_cleanup()`.

Userspace is only allowed to set this to any property value of the given **type**. Only useful for atomic properties, which is enforced.

Return

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

struct *drm_property* ***drm_property_create_bool**(struct *drm_device* *dev, u32 flags, const char *name)

create a new boolean property type

Parameters

struct drm_device *dev

drm device

u32 flags

flags specifying the property type

const char *name

name of the property

Description

This creates a new generic drm property which can then be attached to a drm object with `drm_object_attach_property()`. The returned property object must be freed with `drm_property_destroy()`, which is done automatically when calling `drm_mode_config_cleanup()`.

This is implemented as a ranged property with only {0, 1} as valid values.

Return

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

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

add a possible value to an enumeration property

Parameters

struct drm_property *property

enumeration property to change

uint64_t value

value of the new enumeration

const char *name

symbolic name of the new enumeration

Description

This functions adds enumerations to a property.

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

Return

Zero on success, error code on failure.

void **drm_property_destroy**(struct *drm_device* *dev, struct *drm_property* *property)
destroy a drm property

Parameters

struct drm_device *dev
drm device

struct drm_property *property
property to destroy

Description

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

struct *drm_property_blob* ***drm_property_create_blob**(struct *drm_device* *dev, size_t length, const void *data)

Create new blob property

Parameters

struct drm_device *dev
DRM device to create property for

size_t length
Length to allocate for blob data

const void *data
If specified, copies data into blob

Description

Creates a new blob property for a specified DRM device, optionally copying data. Note that blob properties are meant to be invariant, hence the data must be filled out before the blob is used as the value of any property.

Return

New blob property with a single reference on success, or an ERR_PTR value on failure.

void **drm_property_blob_put**(struct *drm_property_blob* *blob)
release a blob property reference

Parameters

struct drm_property_blob *blob
DRM blob property

Description

Releases a reference to a blob property. May free the object.

struct *drm_property_blob* ***drm_property_blob_get**(struct *drm_property_blob* *blob)
acquire blob property reference

Parameters

struct drm_property_blob *blob
DRM blob property

Description

Acquires a reference to an existing blob property. Returns **blob**, which allows this to be used as a shorthand in assignments.

struct *drm_property_blob* ***drm_property_lookup_blob**(struct *drm_device* *dev, uint32_t id)
look up a blob property and take a reference

Parameters

struct drm_device *dev
drm device

uint32_t id
id of the blob property

Description

If successful, this takes an additional reference to the blob property. callers need to make sure to eventually unreferenced the returned property again, using *drm_property_blob_put()*.

Return

NULL on failure, pointer to the blob on success.

int **drm_property_replace_global_blob**(struct *drm_device* *dev, struct *drm_property_blob* **replace, size_t length, const void *data, struct *drm_mode_object* *obj_holds_id, struct *drm_property* *prop_holds_id)
replace existing blob property

Parameters

struct drm_device *dev
drm device

struct drm_property_blob **replace
location of blob property pointer to be replaced

size_t length
length of data for new blob, or 0 for no data

const void *data
content for new blob, or NULL for no data

struct drm_mode_object *obj_holds_id
optional object for property holding blob ID

struct drm_property *prop_holds_id
optional property holding blob ID **return** 0 on success or error on failure

Description

This function will replace a global property in the blob list, optionally updating a property which holds the ID of that property.

If length is 0 or data is NULL, no new blob will be created, and the holding property, if specified, will be set to 0.

Access to the replace pointer is assumed to be protected by the caller, e.g. by holding the relevant modesetting object lock for its parent.

For example, a `drm_connector` has a 'PATH' property, which contains the ID of a blob property with the value of the MST path information. Calling this function with `replace` pointing to the connector's `path_blob_ptr`, `length` and `data` set for the new path information, `obj_holds_id` set to the connector's base object, and `prop_holds_id` set to the path property name, will perform a completely atomic update. The access to `path_blob_ptr` is protected by the caller holding a lock on the connector.

```
bool drm_property_replace_blob(struct drm_property_blob **blob, struct
                               drm_property_blob *new_blob)
```

replace a blob property

Parameters

```
struct drm_property_blob **blob
```

a pointer to the member blob to be replaced

```
struct drm_property_blob *new_blob
```

the new blob to replace with

Return

true if the blob was in fact replaced.

4.14.3 Standard Connector Properties

DRM connectors have a few standardized properties:

EDID:

Blob property which contains the current EDID read from the sink. This is useful to parse sink identification information like vendor, model and serial. Drivers should update this property by calling `drm_connector_update_edid_property()`, usually after having parsed the EDID using `drm_add_edid_modes()`. Userspace cannot change this property.

User-space should not parse the EDID to obtain information exposed via other KMS properties (because the kernel might apply limits, quirks or fixups to the EDID). For instance, user-space should not try to parse mode lists from the EDID.

DPMS:

Legacy property for setting the power state of the connector. For atomic drivers this is only provided for backwards compatibility with existing drivers, it remaps to controlling the "ACTIVE" property on the CRTC the connector is linked to. Drivers should never set this property directly, it is handled by the DRM core by calling the `drm_connector_funcs.dpms` callback. For atomic drivers the remapping to the "ACTIVE" property is implemented in the DRM core.

Note that this property cannot be set through the `MODE_ATOMIC` ioctl, userspace must use “ACTIVE” on the CRTC instead.

WARNING:

For userspace also running on legacy drivers the “DPMS” semantics are a lot more complicated. First, userspace cannot rely on the “DPMS” value returned by the `GETCONNECTOR` actually reflecting reality, because many drivers fail to update it. For atomic drivers this is taken care of in `drm_atomic_helper_update_legacy_modeset_state()`.

The second issue is that the DPMS state is only well-defined when the connector is connected to a CRTC. In atomic the DRM core enforces that “ACTIVE” is off in such a case, no such checks exists for “DPMS”.

Finally, when enabling an output using the legacy `SETCONFIG` ioctl then “DPMS” is forced to ON. But see above, that might not be reflected in the software value on legacy drivers.

Summarizing: Only set “DPMS” when the connector is known to be enabled, assume that a successful `SETCONFIG` call also sets “DPMS” to on, and never read back the value of “DPMS” because it can be incorrect.

PATH:

Connector path property to identify how this sink is physically connected. Used by DP MST. This should be set by calling `drm_connector_set_path_property()`, in the case of DP MST with the path property the MST manager created. Userspace cannot change this property.

TILE:

Connector tile group property to indicate how a set of DRM connector compose together into one logical screen. This is used by both high-res external screens (often only using a single cable, but exposing multiple DP MST sinks), or high-res integrated panels (like dual-link DSI) which are not gen-locked. Note that for tiled panels which are genlocked, like dual-link LVDS or dual-link DSI, the driver should try to not expose the tiling and virtualise both `drm_crtc` and `drm_plane` if needed. Drivers should update this value using `drm_connector_set_tile_property()`. Userspace cannot change this property.

link-status:

Connector link-status property to indicate the status of link. The default value of link-status is “GOOD”. If something fails during or after modeset, the kernel driver may set this to “BAD” and issue a hotplug uevent. Drivers should update this value using `drm_connector_set_link_status_property()`.

When user-space receives the hotplug uevent and detects a “BAD” link-status, the sink doesn’t receive pixels anymore (e.g. the screen becomes completely black). The list of available modes may have changed. User-space is expected to pick a new mode if the current one has disappeared and perform a new modeset with link-status set to “GOOD” to re-enable the connector.

If multiple connectors share the same CRTC and one of them gets a “BAD” link-status, the other are unaffected (ie. the sinks still continue to receive pixels).

When user-space performs an atomic commit on a connector with a “BAD” link-status without resetting the property to “GOOD”, the sink may still not receive pixels. When user-space performs an atomic commit which resets the link-status property to “GOOD” without the `ALLOW_MODESET` flag set, it might fail because a modeset is required.

User-space can only change link-status to “GOOD”, changing it to “BAD” is a no-op.

For backwards compatibility with non-atomic userspace the kernel tries to automatically set the link-status back to “GOOD” in the SETCRTC IOCTL. This might fail if the mode is no longer valid, similar to how it might fail if a different screen has been connected in the interim.

non_desktop:

Indicates the output should be ignored for purposes of displaying a standard desktop environment or console. This is most likely because the output device is not rectilinear.

Content Protection:

This property is used by userspace to request the kernel protect future content communicated over the link. When requested, kernel will apply the appropriate means of protection (most often HDCP), and use the property to tell userspace the protection is active.

Drivers can set this up by calling `drm_connector_attach_content_protection_property()` on initialization.

The value of this property can be one of the following:

DRM_MODE_CONTENT_PROTECTION_UNDESIRED = 0

The link is not protected, content is transmitted in the clear.

DRM_MODE_CONTENT_PROTECTION_DESIRED = 1

Userspace has requested content protection, but the link is not currently protected. When in this state, kernel should enable Content Protection as soon as possible.

DRM_MODE_CONTENT_PROTECTION_ENABLED = 2

Userspace has requested content protection, and the link is protected. Only the driver can set the property to this value. If userspace attempts to set to ENABLED, kernel will return -EINVAL.

A few guidelines:

- DESIRED state should be preserved until userspace de-asserts it by setting the property to UNDESIRED. This means ENABLED should only transition to UNDESIRED when the user explicitly requests it.
- If the state is DESIRED, kernel should attempt to re-authenticate the link whenever possible. This includes across disable/enable, dpms, hotplug, downstream device changes, link status failures, etc..
- Kernel sends uevent with the connector id and property id through **drm_hdcp_update_content_protection**, upon below kernel triggered scenarios:
 - DESIRED -> ENABLED (authentication success)
 - ENABLED -> DESIRED (termination of authentication)
- Please note no uevents for userspace triggered property state changes, which can't fail such as
 - DESIRED/ENABLED -> UNDESIRED
 - UNDESIRED -> DESIRED
- Userspace is responsible for polling the property or listen to uevents to determine when the value transitions from ENABLED to DESIRED. This signifies the link is no longer protected and userspace should take appropriate action (whatever that might be).

HDCP Content Type:

This Enum property is used by the userspace to declare the content type of the display stream, to kernel. Here display stream stands for any display content that userspace intended to display through HDCP encryption.

Content Type of a stream is decided by the owner of the stream, as “HDCP Type0” or “HDCP Type1”.

The value of the property can be one of the below:

- “HDCP Type0”: `DRM_MODE_HDCP_CONTENT_TYPE0 = 0`
- “HDCP Type1”: `DRM_MODE_HDCP_CONTENT_TYPE1 = 1`

When kernel starts the HDCP authentication (see “Content Protection” for details), it uses the content type in “HDCP Content Type” for performing the HDCP authentication with the display sink.

Please note in HDCP spec versions, a link can be authenticated with HDCP 2.2 for Content Type 0/Content Type 1. Where as a link can be authenticated with HDCP1.4 only for Content Type 0(though it is implicit in nature. As there is no reference for Content Type in HDCP1.4).

HDCP2.2 authentication protocol itself takes the “Content Type” as a parameter, which is a input for the DP HDCP2.2 encryption algo.

In case of Type 0 content protection request, kernel driver can choose either of HDCP spec versions 1.4 and 2.2. When HDCP2.2 is used for “HDCP Type 0”, a HDCP 2.2 capable repeater in the downstream can send that content to a HDCP 1.4 authenticated HDCP sink (Type0 link). But if the content is classified as “HDCP Type 1”, above mentioned HDCP 2.2 repeater wont send the content to the HDCP sink as it can’t authenticate the HDCP1.4 capable sink for “HDCP Type 1”.

Please note userspace can be ignorant of the HDCP versions used by the kernel driver to achieve the “HDCP Content Type”.

At current scenario, classifying a content as Type 1 ensures that the content will be displayed only through the HDCP2.2 encrypted link.

Note that the HDCP Content Type property is introduced at HDCP 2.2, and defaults to type 0. It is only exposed by drivers supporting HDCP 2.2 (hence supporting Type 0 and Type 1). Based on how next versions of HDCP specs are defined content Type could be used for higher versions too.

If content type is changed when “Content Protection” is not UNDESIRE, then kernel will disable the HDCP and re-enable with new type in the same atomic commit. And when “Content Protection” is ENABLED, it means that link is HDCP authenticated and encrypted, for the transmission of the Type of stream mentioned at “HDCP Content Type”.

HDR_OUTPUT_METADATA:

Connector property to enable userspace to send HDR Metadata to driver. This metadata is based on the composition and blending policies decided by user, taking into account the hardware and sink capabilities. The driver gets this metadata and creates a Dynamic Range and Mastering Infoframe (DRM) in case of HDMI, SDP packet (Non-audio INFOFRAME SDP v1.3) for DP. This is then sent to sink. This notifies the sink of the upcoming frame’s Color Encoding and Luminance parameters.

Userspace first need to detect the HDR capabilities of sink by reading and parsing the EDID. Details of HDR metadata for HDMI are added in CTA 861.G spec. For DP , its defined in VESA DP Standard v1.4. It needs to then get the metadata information of the video/game/app content which are encoded in HDR (basically using HDR transfer functions). With this information it needs to decide on a blending policy and compose the relevant layers/overlays into a common format. Once this blending is done, userspace will be aware of the metadata of the composed frame to be send to sink. It then uses this property to communicate this metadata to driver which then make a Infoframe packet and sends to sink based on the type of encoder connected.

Userspace will be responsible to do Tone mapping operation in case:

- Some layers are HDR and others are SDR
- HDR layers luminance is not same as sink

It will even need to do colorspace conversion and get all layers to one common colorspace for blending. It can use either GL, Media or display engine to get this done based on the capabilities of the associated hardware.

Driver expects metadata to be put in `struct hdr_output_metadata` structure from userspace. This is received as blob and stored in `drm_connector_state.hdr_output_metadata`. It parses EDID and saves the sink metadata in `struct hdr_sink_metadata`, as `drm_connector.hdr_sink_metadata`. Driver uses `drm_hdmi_infoframe_set_hdr_metadata()` helper to set the HDR metadata, `hdmi_drm_infoframe_pack()` to pack the infoframe as per spec, in case of HDMI encoder.

max bpc:

This range property is used by userspace to limit the bit depth. When used the driver would limit the bpc in accordance with the valid range supported by the hardware and sink. Drivers to use the function `drm_connector_attach_max_bpc_property()` to create and attach the property to the connector during initialization.

Connectors also have one standardized atomic property:

CRTC_ID:

Mode object ID of the `drm_crtc` this connector should be connected to.

Connectors for LCD panels may also have one standardized property:

panel orientation:

On some devices the LCD panel is mounted in the casing in such a way that the up/top side of the panel does not match with the top side of the device. Userspace can use this property to check for this. Note that input coordinates from touchscreens (input devices with INPUT_PROP_DIRECT) will still map 1:1 to the actual LCD panel coordinates, so if userspace rotates the picture to adjust for the orientation it must also apply the same transformation to the touchscreen input coordinates. This property is initialized by calling `drm_connector_set_panel_orientation()` or `drm_connector_set_panel_orientation_with_quirk()`

scaling mode:

This property defines how a non-native mode is upscaled to the native mode of an LCD panel:

None:

No upscaling happens, scaling is left to the panel. Not all drivers expose this mode.

Full:

The output is upscaled to the full resolution of the panel, ignoring the aspect ratio.

Center:

No upscaling happens, the output is centered within the native resolution the panel.

Full aspect:

The output is upscaled to maximize either the width or height while retaining the aspect ratio.

This property should be set up by calling `drm_connector_attach_scaling_mode_property()`. Note that drivers can also expose this property to external outputs, in which case they must support “None”, which should be the default (since external screens have a built-in scaler).

subconnector:

This property is used by DVI-I, TVout and DisplayPort to indicate different connector subtypes. Enum values more or less match with those from main connector types. For DVI-I and TVout there is also a matching property “select subconnector” allowing to switch between signal types. DP subconnector corresponds to a downstream port.

privacy-screen sw-state, privacy-screen hw-state:

These 2 optional properties can be used to query the state of the electronic privacy screen that is available on some displays; and in some cases also control the state. If a driver implements these properties then both properties must be present.

“privacy-screen hw-state” is read-only and reflects the actual state of the privacy-screen, possible values: “Enabled”, “Disabled”, “Enabled-locked”, “Disabled-locked”. The locked states indicate that the state cannot be changed through the DRM API. E.g. there might be devices where the firmware-setup options, or a hardware slider-switch, offer always on / off modes.

“privacy-screen sw-state” can be set to change the privacy-screen state when not locked. In this case the driver must update the hw-state property to reflect the new state on completion of the commit of the sw-state property. Setting the sw-state property when the hw-state is locked must be interpreted by the driver as a request to change the state to the set state when the hw-state becomes unlocked. E.g. if “privacy-screen hw-state” is “Enabled-locked” and the sw-state gets set to “Disabled” followed by the user unlocking the state by changing the slider-switch position, then the driver must set the state to “Disabled” upon receiving the unlock event.

In some cases the privacy-screen’s actual state might change outside of control of the DRM code. E.g. there might be a firmware handled hotkey which toggles the actual state, or the actual state might be changed through another userspace API such as writing `/proc/acpi/ibm/lcdshadow`. In this case the driver must update both the hw-state and the sw-state to reflect the new value, overwriting any pending state requests in the sw-state. Any pending sw-state requests are thus discarded.

Note that the ability for the state to change outside of control of the DRM master process means that userspace must not cache the value of the sw-state. Caching the sw-state value and including it in later atomic commits may lead to overriding a state change done through e.g. a firmware handled hotkey. Therefore userspace must not include the privacy-screen sw-state in an atomic commit unless it wants to change its value.

left margin, right margin, top margin, bottom margin:

Add margins to the connector’s viewport. This is typically used to mitigate overscan on

TVs.

The value is the size in pixels of the black border which will be added. The attached CRTC's content will be scaled to fill the whole area inside the margin.

The margins configuration might be sent to the sink, e.g. via HDMI AVI InfoFrames.

Drivers can set up these properties by calling `drm_mode_create_tv_margin_properties()`.

Colorspace:

This property helps select a suitable colorspace based on the sink capability. Modern sink devices support wider gamut like BT2020. This helps switch to BT2020 mode if the BT2020 encoded video stream is being played by the user, same for any other colorspace. Thereby giving a good visual experience to users.

The expectation from userspace is that it should parse the EDID and get supported colorspace. Use this property and switch to the one supported. Sink supported colorspace should be retrieved by userspace from EDID and driver will not explicitly expose them.

Basically the expectation from userspace is:

- Set up CRTC DEGAMMA/CTM/GAMMA to convert to some sink colorspace
- Set this new property to let the sink know what it converted the CRTC output to.
- This property is just to inform sink what colorspace source is trying to drive.

Because between HDMI and DP have different colorspaces, `drm_mode_create_hdmi_colorspace_property()` is used for HDMI connector and `drm_mode_create_dp_colorspace_property()` is used for DP connector.

4.14.4 HDMI Specific Connector Properties

content type (HDMI specific):

Indicates content type setting to be used in HDMI infoframes to indicate content type for the external device, so that it adjusts its display settings accordingly.

The value of this property can be one of the following:

No Data:

Content type is unknown

Graphics:

Content type is graphics

Photo:

Content type is photo

Cinema:

Content type is cinema

Game:

Content type is game

The meaning of each content type is defined in CTA-861-G table 15.

Drivers can set up this property by calling `drm_connector_attach_content_type_property()`. Decoding to infoframe values is done through `drm_hdmi_avi_infoframe_content_type()`.

4.14.5 Analog TV Specific Connector Properties

TV Mode:

Indicates the TV Mode used on an analog TV connector. The value of this property can be one of the following:

NTSC:

TV Mode is CCIR System M (aka 525-lines) together with the NTSC Color Encoding.

NTSC-443:

TV Mode is CCIR System M (aka 525-lines) together with the NTSC Color Encoding, but with a color subcarrier frequency of 4.43MHz

NTSC-J:

TV Mode is CCIR System M (aka 525-lines) together with the NTSC Color Encoding, but with a black level equal to the blanking level.

PAL:

TV Mode is CCIR System B (aka 625-lines) together with the PAL Color Encoding.

PAL-M:

TV Mode is CCIR System M (aka 525-lines) together with the PAL Color Encoding.

PAL-N:

TV Mode is CCIR System N together with the PAL Color Encoding, a color subcarrier frequency of 3.58MHz, the SECAM color space, and narrower channels than other PAL variants.

SECAM:

TV Mode is CCIR System B (aka 625-lines) together with the SECAM Color Encoding.

Drivers can set up this property by calling `drm_mode_create_tv_properties()`.

4.14.6 Standard CRTC Properties

DRM CRTC's have a few standardized properties:

ACTIVE:

Atomic property for setting the power state of the CRTC. When set to 1 the CRTC will actively display content. When set to 0 the CRTC will be powered off. There is no expectation that user-space will reset CRTC resources like the mode and planes when setting ACTIVE to 0.

User-space can rely on an ACTIVE change to 1 to never fail an atomic test as long as no other property has changed. If a change to ACTIVE fails an atomic test, this is a driver bug. For this reason setting ACTIVE to 0 must not release internal resources (like reserved memory bandwidth or clock generators).

Note that the legacy DPMS property on connectors is internally routed to control this property for atomic drivers.

MODE_ID:

Atomic property for setting the CRTC display timings. The value is the ID of a blob containing the DRM mode info. To disable the CRTC, user-space must set this property to 0.

Setting MODE_ID to 0 will release reserved resources for the CRTC.

SCALING_FILTER:

Atomic property for setting the scaling filter for CRTC scaler

The value of this property can be one of the following:

Default:

Driver's default scaling filter

Nearest Neighbor:

Nearest Neighbor scaling filter

4.14.7 Standard Plane Properties

DRM planes have a few standardized properties:

type:

Immutable property describing the type of the plane.

For user-space which has enabled the `DRM_CLIENT_CAP_ATOMIC` capability, the plane type is just a hint and is mostly superseded by atomic test-only commits. The type hint can still be used to come up more easily with a plane configuration accepted by the driver.

The value of this property can be one of the following:

“Primary”:

To light up a CRTC, attaching a primary plane is the most likely to work if it covers the whole CRTC and doesn't have scaling or cropping set up.

Drivers may support more features for the primary plane, user-space can find out with test-only atomic commits.

Some primary planes are implicitly used by the kernel in the legacy IOCTLs `DRM_IOCTL_MODE_SETCRTC` and `DRM_IOCTL_MODE_PAGE_FLIP`. Therefore user-space must not mix explicit usage of any primary plane (e.g. through an atomic commit) with these legacy IOCTLs.

“Cursor”:

To enable this plane, using a framebuffer configured without scaling or cropping and with the following properties is the most likely to work:

- If the driver provides the capabilities `DRM_CAP_CURSOR_WIDTH` and `DRM_CAP_CURSOR_HEIGHT`, create the framebuffer with this size. Otherwise, create a framebuffer with the size 64x64.
- If the driver doesn't support modifiers, create a framebuffer with a linear layout. Otherwise, use the `IN_FORMATS` plane property.

Drivers may support more features for the cursor plane, user-space can find out with test-only atomic commits.

Some cursor planes are implicitly used by the kernel in the legacy IOCTLs `DRM_IOCTL_MODE_CURSOR` and `DRM_IOCTL_MODE_CURSOR2`. Therefore user-space must

not mix explicit usage of any cursor plane (e.g. through an atomic commit) with these legacy IOCTLs.

Some drivers may support cursors even if no cursor plane is exposed. In this case, the legacy cursor IOCTLs can be used to configure the cursor.

“Overlay”:

Neither primary nor cursor.

Overlay planes are the only planes exposed when the `DRM_CLIENT_CAP_UNIVERSAL_PLANES` capability is disabled.

IN_FORMATS:

Blob property which contains the set of buffer format and modifier pairs supported by this plane. The blob is a struct `drm_format_modifier_blob`. Without this property the plane doesn't support buffers with modifiers. Userspace cannot change this property.

Note that userspace can check the `DRM_CAP_ADDFB2_MODIFIERS` driver capability for general modifier support. If this flag is set then every plane will have the `IN_FORMATS` property, even when it only supports `DRM_FORMAT_MOD_LINEAR`. Before linux kernel release v5.1 there have been various bugs in this area with inconsistencies between the capability flag and per-plane properties.

4.14.8 Plane Composition Properties

The basic plane composition model supported by standard plane properties only has a source rectangle (in logical pixels within the `drm_framebuffer`), with sub-pixel accuracy, which is scaled up to a pixel-aligned destination rectangle in the visible area of a `drm_crtc`. The visible area of a CRTC is defined by the horizontal and vertical visible pixels (stored in `hdisplay` and `vdisplay`) of the requested mode (stored in `drm_crtc_state.mode`). These two rectangles are both stored in the `drm_plane_state`.

For the atomic ioctl the following standard (atomic) properties on the plane object encode the basic plane composition model:

SRC_X:

X coordinate offset for the source rectangle within the `drm_framebuffer`, in 16.16 fixed point. Must be positive.

SRC_Y:

Y coordinate offset for the source rectangle within the `drm_framebuffer`, in 16.16 fixed point. Must be positive.

SRC_W:

Width for the source rectangle within the `drm_framebuffer`, in 16.16 fixed point. SRC_X plus SRC_W must be within the width of the source framebuffer. Must be positive.

SRC_H:

Height for the source rectangle within the `drm_framebuffer`, in 16.16 fixed point. SRC_Y plus SRC_H must be within the height of the source framebuffer. Must be positive.

CRTC_X:

X coordinate offset for the destination rectangle. Can be negative.

CRTC_Y:

Y coordinate offset for the destination rectangle. Can be negative.

CRTC_W:

Width for the destination rectangle. CRTC_X plus CRTC_W can extend past the currently visible horizontal area of the *drm_crtc*.

CRTC_H:

Height for the destination rectangle. CRTC_Y plus CRTC_H can extend past the currently visible vertical area of the *drm_crtc*.

FB_ID:

Mode object ID of the *drm_framebuffer* this plane should scan out.

CRTC_ID:

Mode object ID of the *drm_crtc* this plane should be connected to.

Note that the source rectangle must fully lie within the bounds of the *drm_framebuffer*. The destination rectangle can lie outside of the visible area of the current mode of the CRTC. It must be appropriately clipped by the driver, which can be done by calling *drm_plane_helper_check_update()*. Drivers are also allowed to round the subpixel sampling positions appropriately, but only to the next full pixel. No pixel outside of the source rectangle may ever be sampled, which is important when applying more sophisticated filtering than just a bilinear one when scaling. The filtering mode when scaling is unspecified.

On top of this basic transformation additional properties can be exposed by the driver:

alpha:

Alpha is setup with *drm_plane_create_alpha_property()*. It controls the plane-wide opacity, from transparent (0) to opaque (0xffff). It can be combined with pixel alpha. The pixel values in the framebuffers are expected to not be pre-multiplied by the global alpha associated to the plane.

rotation:

Rotation is set up with *drm_plane_create_rotation_property()*. It adds a rotation and reflection step between the source and destination rectangles. Without this property the rectangle is only scaled, but not rotated or reflected.

Possible values:

“rotate-<degrees>”:

Signals that a drm plane is rotated <degrees> degrees in counter clockwise direction.

“reflect-<axis>”:

Signals that the contents of a drm plane is reflected along the <axis> axis, in the same way as mirroring.

reflect-x:

o				o
		->		
	v			v

reflect-y:

o				^
		->		
	v			o

zpos:

Z position is set up with `drm_plane_create_zpos_immutable_property()` and `drm_plane_create_zpos_property()`. It controls the visibility of overlapping planes. Without this property the primary plane is always below the cursor plane, and ordering between all other planes is undefined. The positive Z axis points towards the user, i.e. planes with lower Z position values are underneath planes with higher Z position values. Two planes with the same Z position value have undefined ordering. Note that the Z position value can also be immutable, to inform userspace about the hard-coded stacking of planes, see `drm_plane_create_zpos_immutable_property()`. If any plane has a zpos property (either mutable or immutable), then all planes shall have a zpos property.

pixel blend mode:

Pixel blend mode is set up with `drm_plane_create_blend_mode_property()`. It adds a blend mode for alpha blending equation selection, describing how the pixels from the current plane are composited with the background.

Three alpha blending equations are defined:

“None”:

Blend formula that ignores the pixel alpha:

```
out.rgb = plane_alpha * fg.rgb +  
         (1 - plane_alpha) * bg.rgb
```

“Pre-multiplied”:

Blend formula that assumes the pixel color values have been already pre-multiplied with the alpha channel values:

```
out.rgb = plane_alpha * fg.rgb +  
         (1 - (plane_alpha * fg.alpha)) * bg.rgb
```

“Coverage”:

Blend formula that assumes the pixel color values have not been pre-multiplied and will do so when blending them to the background color values:

```
out.rgb = plane_alpha * fg.alpha * fg.rgb +  
         (1 - (plane_alpha * fg.alpha)) * bg.rgb
```

Using the following symbols:

“fg.rgb”:

Each of the RGB component values from the plane’s pixel

“fg.alpha”:

Alpha component value from the plane’s pixel. If the plane’s pixel format has no alpha component, then this is assumed to be 1.0. In these cases, this property has no effect, as all three equations become equivalent.

“bg.rgb”:

Each of the RGB component values from the background

“plane_alpha”:

Plane alpha value set by the plane “alpha” property. If the plane does not expose the “alpha” property, then this is assumed to be 1.0

Note that all the property extensions described here apply either to the plane or the CRTC (e.g. for the background color, which currently is not exposed and assumed to be black).

SCALING_FILTER:

Indicates scaling filter to be used for plane scaler

The value of this property can be one of the following:

Default:

Driver's default scaling filter

Nearest Neighbor:

Nearest Neighbor scaling filter

Drivers can set up this property for a plane by calling `drm_plane_create_scaling_filter_property`

4.14.9 Damage Tracking Properties

`FB_DAMAGE_CLIPS` is an optional plane property which provides a means to specify a list of damage rectangles on a plane in framebuffer coordinates of the framebuffer attached to the plane. In current context damage is the area of plane framebuffer that has changed since last plane update (also called page-flip), irrespective of whether currently attached framebuffer is same as framebuffer attached during last plane update or not.

`FB_DAMAGE_CLIPS` is a hint to kernel which could be helpful for some drivers to optimize internally especially for virtual devices where each framebuffer change needs to be transmitted over network, usb, etc.

Since `FB_DAMAGE_CLIPS` is a hint so it is an optional property. User-space can ignore damage clips property and in that case driver will do a full plane update. In case damage clips are provided then it is guaranteed that the area inside damage clips will be updated to plane. For efficiency driver can do full update or can update more than specified in damage clips. Since driver is free to read more, user-space must always render the entire visible framebuffer. Otherwise there can be corruptions. Also, if a user-space provides damage clips which doesn't encompass the actual damage to framebuffer (since last plane update) can result in incorrect rendering.

`FB_DAMAGE_CLIPS` is a blob property with the layout of blob data is simply an array of `drm_mode_rect`. Unlike plane `drm_plane_state.src` coordinates, damage clips are not in 16.16 fixed point. Similar to plane `src` in framebuffer, damage clips cannot be negative. In damage clip, `x1/y1` are inclusive and `x2/y2` are exclusive. While kernel does not error for overlapped damage clips, it is strongly discouraged.

Drivers that are interested in damage interface for plane should enable `FB_DAMAGE_CLIPS` property by calling `drm_plane_enable_fb_damage_clips()`. Drivers implementing damage can use `drm_atomic_helper_damage_iter_init()` and `drm_atomic_helper_damage_iter_next()` helper iterator function to get damage rectangles clipped to `drm_plane_state.src`.

4.14.10 Color Management Properties

Color management or color space adjustments is supported through a set of 5 properties on the `drm_crtc` object. They are set up by calling `drm_crtc_enable_color_mgmt()`.

“DEGAMMA_LUT”:

Blob property to set the degamma lookup table (LUT) mapping pixel data from the frame-buffer before it is given to the transformation matrix. The data is interpreted as an array of `struct drm_color_lut` elements. Hardware might choose not to use the full precision of the LUT elements nor use all the elements of the LUT (for example the hardware might choose to interpolate between LUT[0] and LUT[4]).

Setting this to NULL (blob property value set to 0) means a linear/pass-thru gamma table should be used. This is generally the driver boot-up state too. Drivers can access this blob through `drm_crtc_state.degamma_lut`.

“DEGAMMA_LUT_SIZE”:

Unsigned range property to give the size of the lookup table to be set on the DEGAMMA_LUT property (the size depends on the underlying hardware). If drivers support multiple LUT sizes then they should publish the largest size, and sub-sample smaller sized LUTs (e.g. for split-gamma modes) appropriately.

“CTM”:

Blob property to set the current transformation matrix (CTM) apply to pixel data after the lookup through the degamma LUT and before the lookup through the gamma LUT. The data is interpreted as a `struct drm_color_ctm`.

Setting this to NULL (blob property value set to 0) means a unit/pass-thru matrix should be used. This is generally the driver boot-up state too. Drivers can access the blob for the color conversion matrix through `drm_crtc_state.ctm`.

“GAMMA_LUT”:

Blob property to set the gamma lookup table (LUT) mapping pixel data after the transformation matrix to data sent to the connector. The data is interpreted as an array of `struct drm_color_lut` elements. Hardware might choose not to use the full precision of the LUT elements nor use all the elements of the LUT (for example the hardware might choose to interpolate between LUT[0] and LUT[4]).

Setting this to NULL (blob property value set to 0) means a linear/pass-thru gamma table should be used. This is generally the driver boot-up state too. Drivers can access this blob through `drm_crtc_state.gamma_lut`.

Note that for mostly historical reasons stemming from Xorg heritage, this is also used to store the color map (also sometimes color lut, CLUT or color palette) for indexed formats like `DRM_FORMAT_C8`.

“GAMMA_LUT_SIZE”:

Unsigned range property to give the size of the lookup table to be set on the GAMMA_LUT property (the size depends on the underlying hardware). If drivers support multiple LUT sizes then they should publish the largest size, and sub-sample smaller sized LUTs (e.g. for split-gamma modes) appropriately.

There is also support for a legacy gamma table, which is set up by calling `drm_mode_crtc_set_gamma_size()`. The DRM core will then alias the legacy gamma ramp with “GAMMA_LUT” or, if that is unavailable, “DEGAMMA_LUT”.

Support for different non RGB color encodings is controlled through *drm_plane* specific COLOR_ENCODING and COLOR_RANGE properties. They are set up by calling *drm_plane_create_color_properties()*.

“COLOR_ENCODING”:

Optional plane enum property to support different non RGB color encodings. The driver can provide a subset of standard enum values supported by the DRM plane.

“COLOR_RANGE”:

Optional plane enum property to support different non RGB color parameter ranges. The driver can provide a subset of standard enum values supported by the DRM plane.

4.14.11 Tile Group Property

Tile groups are used to represent tiled monitors with a unique integer identifier. Tiled monitors using DisplayID v1.3 have a unique 8-byte handle, we store this in a tile group, so we have a common identifier for all tiles in a monitor group. The property is called “TILE”. Drivers can manage tile groups using *drm_mode_create_tile_group()*, *drm_mode_put_tile_group()* and *drm_mode_get_tile_group()*. But this is only needed for internal panels where the tile group information is exposed through a non-standard way.

4.14.12 Explicit Fencing Properties

Explicit fencing allows userspace to control the buffer synchronization between devices. A Fence or a group of fences are transferred to/from userspace using Sync File fds and there are two DRM properties for that. IN_FENCE_FD on each DRM Plane to send fences to the kernel and OUT_FENCE_PTR on each DRM CRTC to receive fences from the kernel.

As a contrast, with implicit fencing the kernel keeps track of any ongoing rendering, and automatically ensures that the atomic update waits for any pending rendering to complete. This is usually tracked in *struct dma_resv* which can also contain mandatory kernel fences. Implicit syncing is how Linux traditionally worked (e.g. DRI2/3 on X.org), whereas explicit fencing is what Android wants.

“IN_FENCE_FD”:

Use this property to pass a fence that DRM should wait on before proceeding with the Atomic Commit request and show the framebuffer for the plane on the screen. The fence can be either a normal fence or a merged one, the sync_file framework will handle both cases and use a fence_array if a merged fence is received. Passing -1 here means no fences to wait on.

If the Atomic Commit request has the DRM_MODE_ATOMIC_TEST_ONLY flag it will only check if the Sync File is a valid one.

On the driver side the fence is stored on the **fence** parameter of *struct drm_plane_state*. Drivers which also support implicit fencing should extract the implicit fence using *drm_gem_plane_helper_prepare_fb()*, to make sure there's consistent behaviour between drivers in precedence of implicit vs. explicit fencing.

“OUT_FENCE_PTR”:

Use this property to pass a file descriptor pointer to DRM. Once the Atomic Commit request call returns OUT_FENCE_PTR will be filled with the file descriptor number of a Sync File.

This Sync File contains the CRTC fence that will be signaled when all framebuffers present on the Atomic Commit * request for that given CRTC are scanned out on the screen.

The Atomic Commit request fails if a invalid pointer is passed. If the Atomic Commit request fails for any other reason the out fence fd returned will be -1. On a Atomic Commit with the `DRM_MODE_ATOMIC_TEST_ONLY` flag the out fence will also be set to -1.

Note that out-fences don't have a special interface to drivers and are internally represented by a `struct drm_pending_vblank_event` in struct `drm_crtc_state`, which is also used by the nonblocking atomic commit helpers and for the DRM event handling for existing userspace.

4.14.13 Variable Refresh Properties

Variable refresh rate capable displays can dynamically adjust their refresh rate by extending the duration of their vertical front porch until page flip or timeout occurs. This can reduce or remove stuttering and latency in scenarios where the page flip does not align with the vblank interval.

An example scenario would be an application flipping at a constant rate of 48Hz on a 60Hz display. The page flip will frequently miss the vblank interval and the same contents will be displayed twice. This can be observed as stuttering for content with motion.

If variable refresh rate was active on a display that supported a variable refresh range from 35Hz to 60Hz no stuttering would be observable for the example scenario. The minimum supported variable refresh rate of 35Hz is below the page flip frequency and the vertical front porch can be extended until the page flip occurs. The vblank interval will be directly aligned to the page flip rate.

Not all userspace content is suitable for use with variable refresh rate. Large and frequent changes in vertical front porch duration may worsen perceived stuttering for input sensitive applications.

Panel brightness will also vary with vertical front porch duration. Some panels may have noticeable differences in brightness between the minimum vertical front porch duration and the maximum vertical front porch duration. Large and frequent changes in vertical front porch duration may produce observable flickering for such panels.

Userspace control for variable refresh rate is supported via properties on the `drm_connector` and `drm_crtc` objects.

“vrr_capable”:

Optional `drm_connector` boolean property that drivers should attach with `drm_connector_attach_vrr_capable_property()` on connectors that could support variable refresh rates. Drivers should update the property value by calling `drm_connector_set_vrr_capable_property()`.

Absence of the property should indicate absence of support.

“VRR_ENABLED”:

Default `drm_crtc` boolean property that notifies the driver that the content on the CRTC is suitable for variable refresh rate presentation. The driver will take this property as a hint to enable variable refresh rate support if the receiver supports it, ie. if the “vrr_capable” property is true on the `drm_connector` object. The vertical front porch duration will be extended until page-flip or timeout when enabled.

The minimum vertical front porch duration is defined as the vertical front porch duration for the current mode.

The maximum vertical front porch duration is greater than or equal to the minimum vertical front porch duration. The duration is derived from the minimum supported variable refresh rate for the connector.

The driver may place further restrictions within these minimum and maximum bounds.

4.14.14 Existing KMS Properties

The following table gives description of drm properties exposed by various modules/drivers. Because this table is very unwieldy, do not add any new properties here. Instead document them in a section above.

Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Description/Restrictions
	DVI-I	"subconnector"	ENUM	{ "Unknown", "DVI-D", "DVI-A" }	Connector	TBD
		"select subconnector"	ENUM	{ "Automatic", "DVI-D", "DVI-A" }	Connector	TBD
	TV	"subconnector"	ENUM	{ "Unknown", "Composite", "SVIDEO", "Component", "SCART" }	Connector	TBD
		"select subconnector"	ENUM	{ "Automatic", "Composite", "SVIDEO", "Component", "SCART" }	Connector	TBD
		"mode"	ENUM	{ "NTSC_M", "NTSC_J", "NTSC_443", "PAL_B" } etc.	Connector	TBD
		"left margin"	RANGE	Min=0, Max=100	Connector	TBD

continues on next page

Table 1 - continued from previous page

Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Description/Restrictions
		"right margin"	RANGE	Min=0, Max=100	Connector	TBD
		"top margin"	RANGE	Min=0, Max=100	Connector	TBD
		"bottom margin"	RANGE	Min=0, Max=100	Connector	TBD
		"brightness"	RANGE	Min=0, Max=100	Connector	TBD
		"contrast"	RANGE	Min=0, Max=100	Connector	TBD
		"flicker reduction"	RANGE	Min=0, Max=100	Connector	TBD
		"overscan"	RANGE	Min=0, Max=100	Connector	TBD
		"saturation"	RANGE	Min=0, Max=100	Connector	TBD
		"hue"	RANGE	Min=0, Max=100	Connector	TBD
	Virtual GPU	"suggested X"	RANGE	Min=0, Max=0xffffffff	Connector	property to suggest an X offset for a connector
		"suggested Y"	RANGE	Min=0, Max=0xffffffff	Connector	property to suggest an Y offset for a connector
	Optional	"aspect ratio"	ENUM	{ "None", "4:3", "16:9" }	Connector	TDB

continues on next page

Table 1 - continued from previous page

Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Descrip- tion/Restrictions
i915	Generic	"Broadcast RGB"	ENUM	{ "Auto- matic", "Full", "Limited 16:235" }	Connector	When this property is set to Limited 16:235 and CTM is set, the hardware will be programmed with the result of the multiplication of CTM by the limited range matrix to ensure the pixels normally in the range 0..1.0 are remapped to the range 16/255..235/255.
		"audio"	ENUM	{ "force- dvi", "off", "auto", "on" }	Connector	TBD
	SDVO-TV	"mode"	ENUM	{ "NTSC_M", "NTSC_J", "NTSC_443" "PAL_B" } etc.	Connector	TBD
		"left_margin"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"right_margin"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD

continues on next page

Table 1 - continued from previous page

Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Descrip- tion/Restrictions
		"top_margin	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"bot- tom_margin'	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"hpos"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"vpos"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"contrast"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"satura- tion"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"hue"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"sharp- ness"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"flicker_filte	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"flicker_filte	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"flicker_filte	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD

continues on next page

Table 1 - continued from previous page

Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Descrip- tion/Restrictions
		"tv_chroma_	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"tv_luma_filt	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"dot_crawl"	RANGE	Min=0, Max=1	Connector	TBD
	SDVO- TV/LVDS	"bright- ness"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
CDV 500	gma-	Generic	"Broadcast RGB"	ENUM	{ "Full", "Limited 16:235" }	Connector TBD
		"Broadcast RGB"	ENUM	{ "off", "auto", "on" }	Connector	TBD
Poulsbo	Generic	"backlight"	RANGE	Min=0, Max=100	Connector	TBD
	SDVO-TV	"mode"	ENUM	{ "NTSC_M", "NTSC_J", "NTSC_443" "PAL_B" } etc.	Connector	TBD
		"left_margin	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"right_margi	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"top_margin	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"bot- tom_margin'	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD

continues on next page

Table 1 - continued from previous page

Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Descrip- tion/Restrictions
		"hpos"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"vpos"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"contrast"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"satura- tion"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"hue"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"sharp- ness"	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"flicker_filt	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"flicker_filt	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"flicker_filt	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"tv_chroma_	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"tv_luma_filt	RANGE	Min=0, Max= SDVO de- pendent	Connector	TBD
		"dot_crawl"	RANGE	Min=0, Max=1	Connector	TBD

continues on next page

Table 1 - continued from previous page

Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Description/Restrictions
	SDVO-TV/LVDS	"brightness"	RANGE	Min=0, Max=SDVO dependent	Connector	TBD
armada	CRTC	"CSC_YUV"	ENUM	{ "Auto" , "CCIR601", "CCIR709" }	CRTC	TBD
		"CSC_RGB"	ENUM	{ "Auto", "Computer system", "Studio" }	CRTC	TBD
	Overlay	"colorkey"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_min"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_max"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_val"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_alpha"	RANGE	Min=0, Max=0xffffffff	Plane	TBD
		"colorkey_mode"	ENUM	{ "disabled", "Y component", "U component", "V component", "RGB", "R component", "G component", "B component" }	Plane	TBD
		"brightness"	RANGE	Min=0, Max=256 + 255	Plane	TBD
		"contrast"	RANGE	Min=0, Max=0x7fff	Plane	TBD
		"saturation"	RANGE	Min=0, Max=0x7fff	Plane	TBD

continues on next page

Table 1 – continued from previous page

Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Description/Restrictions
exynos	CRTC	“mode”	ENUM	{ “nor- mal”, “blank” }	CRTC	TBD
i2c/ch7006_	Generic	“scale”	RANGE	Min=0, Max=2	Connector	TBD
	TV	“mode”	ENUM	{ “PAL”, “PAL- M”, “PAL- N”}, “PAL-Nc” , “PAL-60”, “NTSC-M”, “NTSC-J” }	Connector	TBD
nouveau	NV10 Overlay	“colorkey”	RANGE	Min=0, Max=0x01ff	Plane	TBD
		“contrast”	RANGE	Min=0, Max=8192- 1	Plane	TBD
		“bright- ness”	RANGE	Min=0, Max=1024	Plane	TBD
		“hue”	RANGE	Min=0, Max=359	Plane	TBD
		“satura- tion”	RANGE	Min=0, Max=8192- 1	Plane	TBD
		“iturbt_709”	RANGE	Min=0, Max=1	Plane	TBD
	Nv04 Over- lay	“colorkey”	RANGE	Min=0, Max=0x01ff	Plane	TBD
		“bright- ness”	RANGE	Min=0, Max=1024	Plane	TBD
	Display	“dithering mode”	ENUM	{ “auto”, “off”, “on” }	Connector	TBD
		“dithering depth”	ENUM	{ “auto”, “off”, “on”, “static 2x2”, “dy- namic 2x2”, “tem- poral” }	Connector	TBD
		“under- scan”	ENUM	{ “auto”, “6 bpc”, “8 bpc” }	Connector	TBD

continues on next page

Table 1 - continued from previous page

Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Description/Restrictions
		"under-scan hborder"	RANGE	Min=0, Max=128	Connector	TBD
		"under-scan vborder"	RANGE	Min=0, Max=128	Connector	TBD
		"vibrant hue"	RANGE	Min=0, Max=180	Connector	TBD
		"color vibrance"	RANGE	Min=0, Max=200	Connector	TBD
omap	Generic	"zorder"	RANGE	Min=0, Max=3	CRTC, Plane	TBD
qxl	Generic	"hot-plug_mode_1"	RANGE	Min=0, Max=1	Connector	TBD
radeon	DVI-I	"coherent"	RANGE	Min=0, Max=1	Connector	TBD
	DAC enable load detect	"load detection"	RANGE	Min=0, Max=1	Connector	TBD
	TV Standard	"tv standard"	ENUM	{ "ntsc", "pal", "pal-m", "pal-60", "ntsc-j", "scart-pal", "pal-cn", "se-cam" }	Connector	TBD
	legacy TMDS PLL detect	"tmds_pll"	ENUM	{ "driver", "bios" }	.	TBD
	Underscan	"under-scan"	ENUM	{ "off", "on", "auto" }	Connector	TBD
		"under-scan hborder"	RANGE	Min=0, Max=128	Connector	TBD
		"under-scan vborder"	RANGE	Min=0, Max=128	Connector	TBD
	Audio	"audio"	ENUM	{ "off", "on", "auto" }	Connector	TBD

continues on next page

Table 1 – continued from previous page

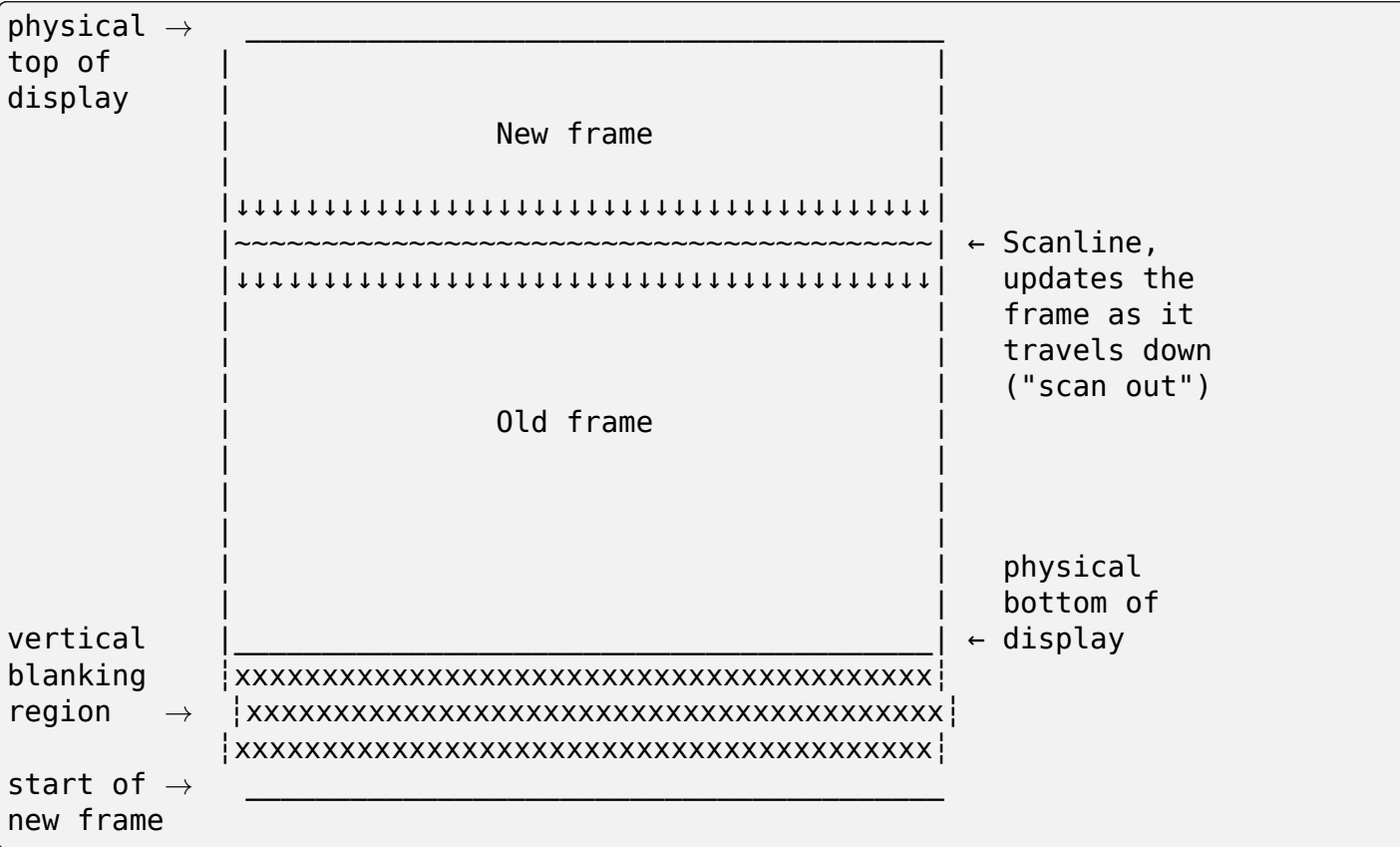
Owner Mod- ule/Drivers	Group	Property Name	Type	Property Values	Object attached	Descrip- tion/Restrictions
	FMT Dither- ing	“dither”	ENUM	{ “off”, “on” }	Connector	TBD
		“colorkey”	RANGE	Min=0, Max=0x01ff	Plane	TBD

4.15 Vertical Blanking

From the computer's perspective, every time the monitor displays a new frame the scanout engine has "scanned out" the display image from top to bottom, one row of pixels at a time. The current row of pixels is referred to as the current scanline.

In addition to the display's visible area, there's usually a couple of extra scanlines which aren't actually displayed on the screen. These extra scanlines don't contain image data and are occasionally used for features like audio and infoframes. The region made up of these scanlines is referred to as the vertical blanking region, or vblank for short.

For historical reference, the vertical blanking period was designed to give the electron gun (on CRTs) enough time to move back to the top of the screen to start scanning out the next frame. Similar for horizontal blanking periods. They were designed to give the electron gun enough time to move back to the other side of the screen to start scanning the next scanline.



“Physical top of display” is the reference point for the high-precision/ corrected timestamp.

On a lot of display hardware, programming needs to take effect during the vertical blanking period so that settings like gamma, the image buffer buffer to be scanned out, etc. can safely be changed without showing any visual artifacts on the screen. In some unforgiving hardware, some of this programming has to both start and end in the same vblank. To help with the timing of the hardware programming, an interrupt is usually available to notify the driver when it can start the updating of registers. The interrupt is in this context named the vblank interrupt.

The vblank interrupt may be fired at different points depending on the hardware. Some hardware implementations will fire the interrupt when the new frame start, other implementations will fire the interrupt at different points in time.

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 initialize the vertical blanking handling core with a call to `drm_vblank_init()`. Minimally, a driver needs to implement `drm_crtc_funcs.enable_vblank` and `drm_crtc_funcs.disable_vblank` plus call `drm_crtc_handle_vblank()` in its vblank interrupt handler for working vblank support.

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_crtc_vblank_get()` and release the vblank reference again with `drm_crtc_vblank_put()`. In between these two calls vblank interrupts are guaranteed to be enabled.

On many hardware disabling the vblank interrupt cannot be done in a race-free manner, see `drm_driver.vblank_disable_immediate` and `drm_driver.max_vblank_count`. In that case the vblank core only disables the vblanks after a timer has expired, which can be configured through the `vblankoffdelay` module parameter.

Drivers for hardware without support for vertical-blanking interrupts must not call `drm_vblank_init()`. For such drivers, atomic helpers will automatically generate fake vblank events as part of the display update. This functionality also can be controlled by the driver by enabling and disabling `struct drm_crtc_state.no_vblank`.

4.15.1 Vertical Blanking and Interrupt Handling Functions Reference

`struct drm_pending_vblank_event`
pending vblank event tracking

Definition:

```
struct drm_pending_vblank_event {
    struct drm_pending_event base;
    unsigned int pipe;
    u64 sequence;
```

```
union {
    struct drm_event base;
    struct drm_event_vblank vbl;
    struct drm_event_crtc_sequence seq;
} event;
};
```

Members

base

Base structure for tracking pending DRM events.

pipe

drm_crtc_index() of the *drm_crtc* this event is for.

sequence

frame event should be triggered at

event

Actual event which will be sent to userspace.

event.base

DRM event base class.

event.vbl

Event payload for vblank events, requested through either the `MODE_PAGE_FLIP` or `MODE_ATOMIC` IOCTL. Also generated by the legacy `WAIT_VBLANK` IOCTL, but new userspace should use `MODE_QUEUE_SEQUENCE` and `event.seq` instead.

event.seq

Event payload for the `MODE_QUEUEU_SEQUENCE` IOCTL.

struct `drm_vblank_crtc`

vblank tracking for a CRTC

Definition:

```
struct drm_vblank_crtc {
    struct drm_device *dev;
    wait_queue_head_t queue;
    struct timer_list disable_timer;
    seqlock_t seqlock;
    atomic64_t count;
    ktime_t time;
    atomic_t refcount;
    u32 last;
    u32 max_vblank_count;
    unsigned int inmodeset;
    unsigned int pipe;
    int framedur_ns;
    int linedur_ns;
    struct drm_display_mode hwmode;
    bool enabled;
    struct kthread_worker *worker;
    struct list_head pending_work;
```

```
wait_queue_head_t work_wait_queue;
};
```

Members

- dev**
Pointer to the *drm_device*.
- queue**
Wait queue for vblank waiters.
- disable_timer**
Disable timer for the delayed vblank disabling hysteresis logic. Vblank disabling is controlled through the *drm_vblank_offdelay* module option and the setting of the *drm_device.max_vblank_count* value.
- seqlock**
Protect vblank count and time.
- count**
Current software vblank counter.
- Note that for a given vblank counter value *drm_crtc_handle_vblank()* and *drm_crtc_vblank_count()* or *drm_crtc_vblank_count_and_time()* provide a barrier: Any writes done before calling *drm_crtc_handle_vblank()* will be visible to callers of the later functions, iff the vblank count is the same or a later one.
- IMPORTANT: This guarantee requires barriers, therefor never access this field directly. Use *drm_crtc_vblank_count()* instead.
- time**
Vblank timestamp corresponding to **count**.
- refcount**
Number of users/waiters of the vblank interrupt. Only when this refcount reaches 0 can the hardware interrupt be disabled using **disable_timer**.
- last**
Protected by *drm_device.vbl_lock*, used for wraparound handling.
- max_vblank_count**
Maximum value of the vblank registers for this crtc. This value +1 will result in a wrap-around of the vblank register. It is used by the vblank core to handle wrap-arounds.
- If set to zero the vblank core will try to guess the elapsed vblanks between times when the vblank interrupt is disabled through high-precision timestamps. That approach is suffering from small races and imprecision over longer time periods, hence exposing a hardware vblank counter is always recommended.
- This is the runtime configurable per-crtc maximum set through *drm_crtc_set_max_vblank_count()*. If this is used the driver must leave the device wide *drm_device.max_vblank_count* at zero.
- If non-zero, *drm_crtc_funcs.get_vblank_counter* must be set.
- inmodeset**
Tracks whether the vblank is disabled due to a modeset. For legacy driver bit 2 additionally tracks whether an additional temporary vblank reference has been acquired to

paper over the hardware counter resetting/jumping. KMS drivers should instead just call `drm_crtc_vblank_off()` and `drm_crtc_vblank_on()`, which explicitly save and restore the vblank count.

pipe

`drm_crtc_index()` of the `drm_crtc` corresponding to this structure.

framedur_ns

Frame/Field duration in ns, used by `drm_crtc_vblank_helper_get_vblank_timestamp()` and computed by `drm_calc_timestamping_constants()`.

linedur_ns

Line duration in ns, used by `drm_crtc_vblank_helper_get_vblank_timestamp()` and computed by `drm_calc_timestamping_constants()`.

hwmode

Cache of the current hardware display mode. Only valid when **enabled** is set. This is used by helpers like `drm_crtc_vblank_helper_get_vblank_timestamp()`. We can't just access the hardware mode by e.g. looking at `drm_crtc_state.adjusted_mode`, because that one is really hard to get from interrupt context.

enabled

Tracks the enabling state of the corresponding `drm_crtc` to avoid double-disabling and hence corrupting saved state. Needed by drivers not using atomic KMS, since those might go through their CRTC disabling functions multiple times.

worker

The `kthread_worker` used for executing vblank works.

pending_work

A list of scheduled `drm_vblank_work` items that are waiting for a future vblank.

work_wait_queue

The wait queue used for signaling that a `drm_vblank_work` item has either finished executing, or was cancelled.

Description

This structure tracks the vblank state for one CRTC.

Note that for historical reasons - the vblank handling code is still shared with legacy/non-kms drivers - this is a free-standing structure not directly connected to `struct drm_crtc`. But all public interface functions are taking a `struct drm_crtc` to hide this implementation detail.

u64 `drm_crtc_accurate_vblank_count`(struct `drm_crtc` *crtc)

retrieve the master vblank counter

Parameters

`struct drm_crtc *crtc`

which counter to retrieve

Description

This function is similar to `drm_crtc_vblank_count()` but this function interpolates to handle a race with vblank interrupts using the high precision timestamping support.

This is mostly useful for hardware that can obtain the scanout position, but doesn't have a hardware frame counter.

int **drm_vblank_init**(struct *drm_device* *dev, unsigned int num_crtcs)
 initialize vblank support

Parameters

struct drm_device *dev
 DRM device

unsigned int num_crtcs
 number of CRTCs supported by **dev**

Description

This function initializes vblank support for **num_crtcs** display pipelines. Cleanup is handled automatically through a cleanup function added with *drmm_add_action_or_reset()*.

Return

Zero on success or a negative error code on failure.

bool **drm_dev_has_vblank**(const struct *drm_device* *dev)
 test if vblanking has been initialized for a device

Parameters

const struct drm_device *dev
 the device

Description

Drivers may call this function to test if vblank support is initialized for a device. For most hardware this means that vblanking can also be enabled.

Atomic helpers use this function to initialize *drm_crtc_state.no_vblank*. See also *drm_atomic_helper_check_modeset()*.

Return

True if vblanking has been initialized for the given device, false otherwise.

wait_queue_head_t ***drm_crtc_vblank_waitqueue**(struct *drm_crtc* *crtc)
 get vblank waitqueue for the CRTC

Parameters

struct drm_crtc *crtc
 which CRTC's vblank waitqueue to retrieve

Description

This function returns a pointer to the vblank waitqueue for the CRTC. Drivers can use this to implement vblank waits using *wait_event()* and related functions.

void **drm_calc_timestamping_constants**(struct *drm_crtc* *crtc, const struct *drm_display_mode* *mode)
 calculate vblank timestamp constants

Parameters

struct drm_crtc *crtc
 drm_crtc whose timestamp constants should be updated.

const struct drm_display_mode *mode
display mode containing the scanout timings

Description

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

bool drm_crtc_vblank_helper_get_vblank_timestamp_internal(struct [drm_crtc](#) *crtc, int *max_error, ktime_t *vblank_time, bool in_vblank_irq, drm_vblank_get_scanout_position_func get_scanout_position)

precise vblank timestamp helper

Parameters

struct drm_crtc *crtc
CRTC whose vblank timestamp to retrieve

int *max_error
Desired maximum allowable error in timestamps (nanosecs) On return contains true maximum error of timestamp

ktime_t *vblank_time
Pointer to time which should receive the timestamp

bool in_vblank_irq
True when called from [drm_crtc_handle_vblank\(\)](#). Some drivers need to apply some workarounds for gpu-specific vblank irq quirks if flag is set.

drm_vblank_get_scanout_position_func get_scanout_position
Callback function to retrieve the scanout position. See **struct** [drm_crtc_helper_funcs.get_scanout_position](#).

Description

Implements calculation of exact vblank timestamps from given [drm_display_mode](#) timings and current video scanout position of a CRTC.

The current implementation only handles standard video modes. For double scan and interlaced modes the driver is supposed to adjust the hardware mode (taken from [drm_crtc_state.adjusted](#) mode for atomic modeset drivers) to match the scanout position reported.

Note that atomic drivers must call [drm_calc_timestamping_constants\(\)](#) before enabling a CRTC. The atomic helpers already take care of that in [drm_atomic_helper_calc_timestamping_constants\(\)](#).

Returns true on success, and false on failure, i.e. when no accurate timestamp could be acquired.

Return

bool drm_crtc_vblank_helper_get_vblank_timestamp(struct [drm_crtc](#) *crtc, int *max_error, ktime_t *vblank_time, bool in_vblank_irq)

precise vblank timestamp helper

Parameters

struct drm_crtc *crtc

CRTC whose vblank timestamp to retrieve

int *max_error

Desired maximum allowable error in timestamps (nanosecs) On return contains true maximum error of timestamp

mtime_t *vblank_time

Pointer to time which should receive the timestamp

bool in_vblank_irq

True when called from [drm_crtc_handle_vblank\(\)](#). Some drivers need to apply some workarounds for gpu-specific vblank irq quirks if flag is set.

Description

Implements calculation of exact vblank timestamps from given `drm_display_mode` timings and current video scanout position of a CRTC. This can be directly used as the [drm_crtc_funcs.get_vblank_timestamp](#) implementation of a kms driver if [drm_crtc_helper_funcs.get_scanout_position](#) is implemented.

The current implementation only handles standard video modes. For double scan and interlaced modes the driver is supposed to adjust the hardware mode (taken from [drm_crtc_state.adjusted](#) mode for atomic modeset drivers) to match the scanout position reported.

Note that atomic drivers must call [drm_calc_timestamping_constants\(\)](#) before enabling a CRTC. The atomic helpers already take care of that in [drm_atomic_helper_calc_timestamping_constants\(\)](#).

Returns true on success, and false on failure, i.e. when no accurate timestamp could be acquired.

Return

u64 **drm_crtc_vblank_count**(struct [drm_crtc](#) *crtc)

retrieve “cooked” vblank counter value

Parameters

struct drm_crtc *crtc

which counter to retrieve

Description

Fetches the “cooked” vblank count value that represents the number of vblank events since the system was booted, including lost events due to modesetting activity. Note that this timer isn’t correct against a racing vblank interrupt (since it only reports the software vblank counter), see [drm_crtc_accurate_vblank_count\(\)](#) for such use-cases.

Note that for a given vblank counter value [drm_crtc_handle_vblank\(\)](#) and [drm_crtc_vblank_count\(\)](#) or [drm_crtc_vblank_count_and_time\(\)](#) provide a barrier: Any writes done before calling [drm_crtc_handle_vblank\(\)](#) will be visible to callers of the later functions, if the vblank count is the same or a later one.

See also [drm_vblank_crtc.count](#).

Return

The software vblank counter.

u64 **drm_crtc_vblank_count_and_time**(struct *drm_crtc* *crtc, ktime_t *vblanktime)
retrieve “cooked” vblank counter value and the system timestamp corresponding to that vblank counter value

Parameters

struct drm_crtc *crtc
which counter to retrieve

ktime_t *vblanktime
Pointer to time to receive the vblank timestamp.

Description

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

Note that for a given vblank counter value *drm_crtc_handle_vblank()* and *drm_crtc_vblank_count()* or *drm_crtc_vblank_count_and_time()* provide a barrier: Any writes done before calling *drm_crtc_handle_vblank()* will be visible to callers of the later functions, if the vblank count is the same or a later one.

See also *drm_vblank_crtc.count*.

int **drm_crtc_next_vblank_start**(struct *drm_crtc* *crtc, ktime_t *vblanktime)
calculate the time of the next vblank

Parameters

struct drm_crtc *crtc
the crtc for which to calculate next vblank time

ktime_t *vblanktime
pointer to time to receive the next vblank timestamp.

Description

Calculate the expected time of the start of the next vblank period, based on time of previous vblank and frame duration

void **drm_crtc_arm_vblank_event**(struct *drm_crtc* *crtc, struct *drm_pending_vblank_event* *e)
arm vblank event after pageflip

Parameters

struct drm_crtc *crtc
the source CRTC of the vblank event

struct drm_pending_vblank_event *e
the event to send

Description

A lot of drivers need to generate vblank events for the very next vblank interrupt. For example when the page flip interrupt happens when the page flip gets armed, but not when it actually executes within the next vblank period. This helper function implements exactly the required vblank arming behaviour.

1. Driver commits new hardware state into vblank-synchronized registers.
2. A vblank happens, committing the hardware state. Also the corresponding vblank interrupt is fired off and fully processed by the interrupt handler.
3. The atomic commit operation proceeds to call `drm_crtc_arm_vblank_event()`.
4. The event is only send out for the next vblank, which is wrong.

An equivalent race can happen when the driver calls `drm_crtc_arm_vblank_event()` before writing out the new hardware state.

The only way to make this work safely is to prevent the vblank from firing (and the hardware from committing anything else) until the entire atomic commit sequence has run to completion. If the hardware does not have such a feature (e.g. using a “go” bit), then it is unsafe to use this functions. Instead drivers need to manually send out the event from their interrupt handler by calling `drm_crtc_send_vblank_event()` and make sure that there’s no possible race with the hardware committing the atomic update.

Caller must hold a vblank reference for the event `e` acquired by a `drm_crtc_vblank_get()`, which will be dropped when the next vblank arrives.

NOTE

Drivers using this to send out the `drm_crtc_state.event` as part of an atomic commit must ensure that the next vblank happens at exactly the same time as the atomic commit is committed to the hardware. This function itself does **not** protect against the next vblank interrupt racing with either this function call or the atomic commit operation. A possible sequence could be:

```
void drm_crtc_send_vblank_event(struct drm_crtc *crtc, struct drm_pending_vblank_event
                               *e)
```

helper to send vblank event after pageflip

Parameters

struct drm_crtc *crtc
the source CRTC of the vblank event

struct drm_pending_vblank_event *e
the event to send

Description

Updates sequence # and timestamp on event for the most recently processed vblank, and sends it to userspace. Caller must hold event lock.

See `drm_crtc_arm_vblank_event()` for a helper which can be used in certain situation, especially to send out events for atomic commit operations.

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

get a reference count on vblank events

Parameters

struct drm_crtc *crtc
which CRTC to own

Description

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

Return

Zero on success or a negative error code on failure.

void **drm_crtc_vblank_put**(struct *drm_crtc* *crtc)
give up ownership of vblank events

Parameters

struct drm_crtc *crtc
which counter to give up

Description

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

void **drm_wait_one_vblank**(struct *drm_device* *dev, unsigned int pipe)
wait for one vblank

Parameters

struct drm_device *dev
DRM device

unsigned int pipe
CRTC index

Description

This waits for one vblank to pass on **pipe**, using the irq driver interfaces. It is a failure to call this when the vblank irq for **pipe** is disabled, e.g. due to lack of driver support or because the crtc is off.

This is the legacy version of *drm_crtc_wait_one_vblank()*.

void **drm_crtc_wait_one_vblank**(struct *drm_crtc* *crtc)
wait for one vblank

Parameters

struct drm_crtc *crtc
DRM crtc

Description

This waits for one vblank to pass on **crtc**, using the irq driver interfaces. It is a failure to call this when the vblank irq for **crtc** is disabled, e.g. due to lack of driver support or because the crtc is off.

void **drm_crtc_vblank_off**(struct *drm_crtc* *crtc)
disable vblank events on a CRTC

Parameters

struct drm_crtc *crtc

CRTC in question

Description

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

Drivers must use this function when the hardware vblank counter can get reset, e.g. when suspending or disabling the **crtc** in general.

void **drm_crtc_vblank_reset**(struct *drm_crtc* *crtc)

reset vblank state to off on a CRTC

Parameters

struct drm_crtc *crtc

CRTC in question

Description

Drivers can use this function to reset the vblank state to off at load time. Drivers should use this together with the *drm_crtc_vblank_off()* and *drm_crtc_vblank_on()* functions. The difference compared to *drm_crtc_vblank_off()* is that this function doesn't save the vblank counter and hence doesn't need to call any driver hooks.

This is useful for recovering driver state e.g. on driver load, or on resume.

void **drm_crtc_set_max_vblank_count**(struct *drm_crtc* *crtc, u32 max_vblank_count)

configure the hw max vblank counter value

Parameters

struct drm_crtc *crtc

CRTC in question

u32 max_vblank_count

max hardware vblank counter value

Description

Update the maximum hardware vblank counter value for **crtc** at runtime. Useful for hardware where the operation of the hardware vblank counter depends on the currently active display configuration.

For example, if the hardware vblank counter does not work when a specific connector is active the maximum can be set to zero. And when that specific connector isn't active the maximum can again be set to the appropriate non-zero value.

If used, must be called before `drm_vblank_on()`.

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

enable vblank events on a CRTC

Parameters

struct drm_crtc *crtc

CRTC in question

Description

This functions restores the vblank interrupt state captured with `drm_crtc_vblank_off()` again and is generally called when enabling **crtc**. Note that calls to `drm_crtc_vblank_on()` and `drm_crtc_vblank_off()` can be unbalanced and so can also be unconditionally called in driver load code to reflect the current hardware state of the crtc.

```
void drm_crtc_vblank_restore(struct drm_crtc *crtc)
    estimate missed vblanks and update vblank count.
```

Parameters

```
struct drm_crtc *crtc
    CRTC in question
```

Description

Power manamement features can cause frame counter resets between vblank disable and enable. Drivers can use this function in their `drm_crtc_funcs.enable_vblank` implementation to estimate missed vblanks since the last `drm_crtc_funcs.disable_vblank` using timestamps and update the vblank counter.

Note that drivers must have race-free high-precision timestamping support, i.e. `drm_crtc_funcs.get_vblank_timestamp` must be hooked up and `drm_driver.vblank_disable_immediate` must be set to indicate the time-stamping functions are race-free against vblank hardware counter increments.

```
bool drm_handle_vblank(struct drm_device *dev, unsigned int pipe)
    handle a vblank event
```

Parameters

```
struct drm_device *dev
    DRM device
```

```
unsigned int pipe
    index of CRTC where this event occurred
```

Description

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

This is the legacy version of `drm_crtc_handle_vblank()`.

```
bool drm_crtc_handle_vblank(struct drm_crtc *crtc)
    handle a vblank event
```

Parameters

```
struct drm_crtc *crtc
    where this event occurred
```

Description

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

This is the native KMS version of `drm_handle_vblank()`.

Note that for a given vblank counter value `drm_crtc_handle_vblank()` and `drm_crtc_vblank_count()` or `drm_crtc_vblank_count_and_time()` provide a barrier: Any writes done before calling `drm_crtc_handle_vblank()` will be visible to callers of the later functions, if the vblank count is the same or a later one.

See also `drm_vblank_crtc.count`.

Return

True if the event was successfully handled, false on failure.

4.16 Vertical Blank Work

Many DRM drivers need to program hardware in a time-sensitive manner, many times with a deadline of starting and finishing within a certain region of the scanout. Most of the time the safest way to accomplish this is to simply do said time-sensitive programming in the driver's IRQ handler, which allows drivers to avoid being preempted during these critical regions. Or even better, the hardware may even handle applying such time-critical programming independently of the CPU.

While there's a decent amount of hardware that's designed so that the CPU doesn't need to be concerned with extremely time-sensitive programming, there's a few situations where it can't be helped. Some unforgiving hardware may require that certain time-sensitive programming be handled completely by the CPU, and said programming may even take too long to handle in an IRQ handler. Another such situation would be where the driver needs to perform a task that needs to complete within a specific scanout period, but might possibly block and thus cannot be handled in an IRQ context. Both of these situations can't be solved perfectly in Linux since we're not a realtime kernel, and thus the scheduler may cause us to miss our deadline if it decides to preempt us. But for some drivers, it's good enough if we can lower our chance of being preempted to an absolute minimum.

This is where `drm_vblank_work` comes in. `drm_vblank_work` provides a simple generic delayed work implementation which delays work execution until a particular vblank has passed, and then executes the work at realtime priority. This provides the best possible chance at performing time-sensitive hardware programming on time, even when the system is under heavy load. `drm_vblank_work` also supports rescheduling, so that self re-arming work items can be easily implemented.

4.16.1 Vertical Blank Work Functions Reference

struct `drm_vblank_work`

A delayed work item which delays until a target vblank passes, and then executes at real-time priority outside of IRQ context.

Definition:

```
struct drm_vblank_work {
    struct kthread_work base;
    struct drm_vblank_crtc *vblank;
    u64 count;
    int cancelling;
```

```
struct list_head node;
};
```

Members

base

The base `kthread_work` item which will be executed by `drm_vblank_crtc.worker`. Drivers should not interact with this directly, and instead rely on `drm_vblank_work_init()` to initialize this.

vblank

A pointer to `drm_vblank_crtc` this work item belongs to.

count

The target vblank this work will execute on. Drivers should not modify this value directly, and instead use `drm_vblank_work_schedule()`

cancelling

The number of `drm_vblank_work_cancel_sync()` calls that are currently running. A work item cannot be rescheduled until all calls have finished.

node

The position of this work item in `drm_vblank_crtc.pending_work`.

Description

See also: `drm_vblank_work_schedule()` `drm_vblank_work_init()`
`drm_vblank_work_cancel_sync()` `drm_vblank_work_flush()`

to_drm_vblank_work

`to_drm_vblank_work (_work)`

Retrieve the respective `drm_vblank_work` item from a `kthread_work`

Parameters

_work

The `kthread_work` embedded inside a `drm_vblank_work`

int **drm_vblank_work_schedule**(struct `drm_vblank_work` *work, u64 count, bool nextonmiss)
schedule a vblank work

Parameters

struct **drm_vblank_work** *work
vblank work to schedule

u64 **count**
target vblank count

bool **nextonmiss**
defer until the next vblank if target vblank was missed

Description

Schedule **work** for execution once the crtc vblank count reaches **count**.

If the crtc vblank count has already reached **count** and **nextonmiss** is false the work starts to execute immediately.

If the crtc vblank count has already reached **count** and **nextonmiss** is true the work is deferred until the next vblank (as if **count** has been specified as crtc vblank count + 1).

If **work** is already scheduled, this function will reschedule said work using the new **count**. This can be used for self-rearming work items.

Return

1 if **work** was successfully (re)scheduled, 0 if it was either already scheduled or cancelled, or a negative error code on failure.

bool **drm_vblank_work_cancel_sync**(struct *drm_vblank_work* *work)

cancel a vblank work and wait for it to finish executing

Parameters

struct *drm_vblank_work* *work

vblank work to cancel

Description

Cancel an already scheduled vblank work and wait for its execution to finish.

On return, **work** is guaranteed to no longer be scheduled or running, even if it's self-arming.

Return

True if the work was cancelled before it started to execute, false otherwise.

void **drm_vblank_work_flush**(struct *drm_vblank_work* *work)

wait for a scheduled vblank work to finish executing

Parameters

struct *drm_vblank_work* *work

vblank work to flush

Description

Wait until **work** has finished executing once.

void **drm_vblank_work_init**(struct *drm_vblank_work* *work, struct *drm_crtc* *crtc, void (*func)(struct kthread_work *work))

initialize a vblank work item

Parameters

struct *drm_vblank_work* *work

vblank work item

struct *drm_crtc* *crtc

CRTC whose vblank will trigger the work execution

void (*func)(struct kthread_work *work)

work function to be executed

Description

Initialize a vblank work item for a specific crtc.

MODE SETTING HELPER FUNCTIONS

The DRM subsystem aims for a strong separation between core code and helper libraries. Core code takes care of general setup and teardown and decoding userspace requests to kernel internal objects. Everything else is handled by a large set of helper libraries, which can be combined freely to pick and choose for each driver what fits, and avoid shared code where special behaviour is needed.

This distinction between core code and helpers is especially strong in the modesetting code, where there's a shared userspace ABI for all drivers. This is in contrast to the render side, where pretty much everything (with very few exceptions) can be considered optional helper code.

There are a few areas these helpers can be grouped into:

- Helpers to implement modesetting. The important ones here are the atomic helpers. Old drivers still often use the legacy CRTC helpers. They both share the same set of common helper vtables. For really simple drivers (anything that would have been a great fit in the deprecated fbdev subsystem) there's also the simple display pipe helpers.
- There's a big pile of helpers for handling outputs. First the generic bridge helpers for handling encoder and transcoder IP blocks. Second the panel helpers for handling panel-related information and logic. Plus then a big set of helpers for the various sink standards (DisplayPort, HDMI, MIPI DSI). Finally there's also generic helpers for handling output probing, and for dealing with EDIDs.
- The last group of helpers concerns itself with the frontend side of a display pipeline: Planes, handling rectangles for visibility checking and scissoring, flip queues and assorted bits.

5.1 Modeset Helper Reference for Common Vtables

The DRM mode setting helper functions are common code for drivers to use if they wish. Drivers are not forced to use this code in their implementations but it would be useful if the code they do use at least provides a consistent interface and operation to userspace. Therefore it is highly recommended to use the provided helpers as much as possible.

Because there is only one pointer per modeset object to hold a vfunc table for helper libraries they are by necessity shared among the different helpers.

To make this clear all the helper vtables are pulled together in this location here.

```
struct drm_crtc_helper_funcs  
    helper operations for CRTCs
```

Definition:

```

struct drm_crtc_helper_funcs {
    void (*dpms)(struct drm_crtc *crtc, int mode);
    void (*prepare)(struct drm_crtc *crtc);
    void (*commit)(struct drm_crtc *crtc);
    enum drm_mode_status (*mode_valid)(struct drm_crtc *crtc, const struct drm_
↪display_mode *mode);
    bool (*mode_fixup)(struct drm_crtc *crtc, const struct drm_display_mode_
↪*mode, struct drm_display_mode *adjusted_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);
    void (*mode_set_nofb)(struct drm_crtc *crtc);
    int (*mode_set_base)(struct drm_crtc *crtc, int x, int y, struct drm_
↪framebuffer *old_fb);
    int (*mode_set_base_atomic)(struct drm_crtc *crtc, struct drm_framebuffer_
↪*fb, int x, int y, enum mode_set_atomic);
    void (*disable)(struct drm_crtc *crtc);
    int (*atomic_check)(struct drm_crtc *crtc, struct drm_atomic_state *state);
    void (*atomic_begin)(struct drm_crtc *crtc, struct drm_atomic_state_
↪*state);
    void (*atomic_flush)(struct drm_crtc *crtc, struct drm_atomic_state_
↪*state);
    void (*atomic_enable)(struct drm_crtc *crtc, struct drm_atomic_state_
↪*state);
    void (*atomic_disable)(struct drm_crtc *crtc, struct drm_atomic_state_
↪*state);
    bool (*get_scanout_position)(struct drm_crtc *crtc, bool in_vblank_irq, int_
↪*vpos, int *hpos, ktime_t *stime, ktime_t *etime, const struct drm_display_
↪mode *mode);
};

```

Members**dpms**

Callback to control power levels on the CRTC. If the mode passed in is unsupported, the provider must use the next lowest power level. This is used by the legacy CRTC helpers to implement DPMS functionality in [drm_helper_connector_dpms\(\)](#).

This callback is also used to disable a CRTC by calling it with `DRM_MODE_DPMS_OFF` if the **disable** hook isn't used.

This callback is used by the legacy CRTC helpers. Atomic helpers also support using this hook for enabling and disabling a CRTC to facilitate transitions to atomic, but it is deprecated. Instead **atomic_enable** and **atomic_disable** should be used.

prepare

This callback should prepare the CRTC for a subsequent modeset, which in practice means the driver should disable the CRTC if it is running. Most drivers ended up implementing this by calling their **dpms** hook with `DRM_MODE_DPMS_OFF`.

This callback is used by the legacy CRTC helpers. Atomic helpers also support using this hook for disabling a CRTC to facilitate transitions to atomic, but it is deprecated. Instead

atomic_disable should be used.

commit

This callback should commit the new mode on the CRTC after a modeset, which in practice means the driver should enable the CRTC. Most drivers ended up implementing this by calling their **dpms** hook with `DRM_MODE_DPMS_ON`.

This callback is used by the legacy CRTC helpers. Atomic helpers also support using this hook for enabling a CRTC to facilitate transitions to atomic, but it is deprecated. Instead **atomic_enable** should be used.

mode_valid

This callback is used to check if a specific mode is valid in this crtc. This should be implemented if the crtc has some sort of restriction in the modes it can display. For example, a given crtc may be responsible to set a clock value. If the clock can not produce all the values for the available modes then this callback can be used to restrict the number of modes to only the ones that can be displayed.

This hook is used by the probe helpers to filter the mode list in [`drm_helper_probe_single_connector_modes\(\)`](#), and it is used by the atomic helpers to validate modes supplied by userspace in [`drm_atomic_helper_check_modeset\(\)`](#).

This function is optional.

NOTE:

Since this function is both called from the check phase of an atomic commit, and the mode validation in the probe paths it is not allowed to look at anything else but the passed-in mode, and validate it against configuration-invariant hardware constraints. Any further limits which depend upon the configuration can only be checked in **mode_fixup** or **atomic_check**.

RETURNS:

`drm_mode_status` Enum

mode_fixup

This callback is used to validate a mode. The parameter `mode` is the display mode that userspace requested, `adjusted_mode` is the mode the encoders need to be fed with. Note that this is the inverse semantics of the meaning for the [`drm_encoder`](#) and [`drm_bridge_funcs.mode_fixup`](#) vfunc. If the CRTC cannot support the requested conversion from `mode` to `adjusted_mode` it should reject the modeset. See also [`drm_crtc_state.adjusted_mode`](#) for more details.

This function is used by both legacy CRTC helpers and atomic helpers. With atomic helpers it is optional.

NOTE:

This function is called in the check phase of atomic modesets, which can be aborted for any reason (including on userspace's request to just check whether a configuration would be possible). Atomic drivers MUST NOT touch any persistent state (hardware or software) or data structures except the passed in `adjusted_mode` parameter.

This is in contrast to the legacy CRTC helpers where this was allowed.

Atomic drivers which need to inspect and adjust more state should instead use the **atomic_check** callback, but note that they're not perfectly equivalent: **mode_valid** is called from [`drm_atomic_helper_check_modeset\(\)`](#), but **atomic_check** is called from

`drm_atomic_helper_check_planes()`, because originally it was meant for plane update checks only.

Also beware that userspace can request its own custom modes, neither core nor helpers filter modes to the list of probe modes reported by the `GETCONNECTOR` IOCTL and stored in `drm_connector.modes`. To ensure that modes are filtered consistently put any CRTC constraints and limits checks into **mode_valid**.

RETURNS:

True if an acceptable configuration is possible, false if the modeset operation should be rejected.

mode_set

This callback is used by the legacy CRTC helpers to set a new mode, position and framebuffer. Since it ties the primary plane to every mode change it is incompatible with universal plane support. And since it can't update other planes it's incompatible with atomic modeset support.

This callback is only used by CRTC helpers and deprecated.

RETURNS:

0 on success or a negative error code on failure.

mode_set_nofb

This callback is used to update the display mode of a CRTC without changing anything of the primary plane configuration. This fits the requirement of atomic and hence is used by the atomic helpers.

Note that the display pipe is completely off when this function is called. Atomic drivers which need hardware to be running before they program the new display mode (e.g. because they implement runtime PM) should not use this hook. This is because the helper library calls this hook only once per mode change and not every time the display pipeline is suspended using either DPMS or the new "ACTIVE" property. Which means register values set in this callback might get reset when the CRTC is suspended, but not restored. Such drivers should instead move all their CRTC setup into the **atomic_enable** callback.

This callback is optional.

mode_set_base

This callback is used by the legacy CRTC helpers to set a new framebuffer and scanout position. It is optional and used as an optimized fast-path instead of a full mode set operation with all the resulting flickering. If it is not present `drm_crtc_helper_set_config()` will fall back to a full modeset, using the **mode_set** callback. Since it can't update other planes it's incompatible with atomic modeset support.

This callback is only used by the CRTC helpers and deprecated.

RETURNS:

0 on success or a negative error code on failure.

mode_set_base_atomic

This callback is used by the fbdev helpers to set a new framebuffer and scanout without sleeping, i.e. from an atomic calling context. It is only used to implement kgdb support.

This callback is optional and only needed for kgdb support in the fbdev helpers.

RETURNS:

0 on success or a negative error code on failure.

disable

This callback should be used to disable the CRTC. With the atomic drivers it is called after all encoders connected to this CRTC have been shut off already using their own [drm_encoder_helper_funcs.disable](#) hook. If that sequence is too simple drivers can just add their own hooks and call it from this CRTC callback here by looping over all encoders connected to it using `for_each_encoder_on_crtc()`.

This hook is used both by legacy CRTC helpers and atomic helpers. Atomic drivers don't need to implement it if there's no need to disable anything at the CRTC level. To ensure that runtime PM handling (using either DPMS or the new "ACTIVE" property) works **disable** must be the inverse of **atomic_enable** for atomic drivers. Atomic drivers should consider to use **atomic_disable** instead of this one.

NOTE:

With legacy CRTC helpers there's a big semantic difference between **disable** and other hooks (like **prepare** or **dpms**) used to shut down a CRTC: **disable** is only called when also logically disabling the display pipeline and needs to release any resources acquired in **mode_set** (like shared PLLs, or again release pinned framebuffers).

Therefore **disable** must be the inverse of **mode_set** plus **commit** for drivers still using legacy CRTC helpers, which is different from the rules under atomic.

atomic_check

Drivers should check plane-update related CRTC constraints in this hook. They can also check mode related limitations but need to be aware of the calling order, since this hook is used by [drm_atomic_helper_check_planes\(\)](#) whereas the preparations needed to check output routing and the display mode is done in [drm_atomic_helper_check_modeset\(\)](#). Therefore drivers that want to check output routing and display mode constraints in this callback must ensure that [drm_atomic_helper_check_modeset\(\)](#) has been called beforehand. This is calling order used by the default helper implementation in [drm_atomic_helper_check\(\)](#).

When using [drm_atomic_helper_check_planes\(\)](#) this hook is called after the [drm_plane_helper_funcs.atomic_check](#) hook for planes, which allows drivers to assign shared resources requested by planes in this callback here. For more complicated dependencies the driver can call the provided check helpers multiple times until the computed state has a final configuration and everything has been checked.

This function is also allowed to inspect any other object's state and can add more state objects to the atomic commit if needed. Care must be taken though to ensure that state check and compute functions for these added states are all called, and derived state in other objects all updated. Again the recommendation is to just call check helpers until a maximal configuration is reached.

This callback is used by the atomic modeset helpers, but it is optional.

NOTE:

This function is called in the check phase of an atomic update. The driver is not allowed to change anything outside of the free-standing state object passed-in.

Also beware that userspace can request its own custom modes, neither core nor helpers filter modes to the list of probe modes reported by the GETCONNECTOR IOCTL and stored

in [`drm_connector.modes`](#). To ensure that modes are filtered consistently put any CRTC constraints and limits checks into **`mode_valid`**.

RETURNS:

0 on success, -EINVAL if the state or the transition can't be supported, -ENOMEM on memory allocation failure and -EDEADLK if an attempt to obtain another state object ran into a [`drm_modeset_lock`](#) deadlock.

`atomic_begin`

Drivers should prepare for an atomic update of multiple planes on a CRTC in this hook. Depending upon hardware this might be vblank evasion, blocking updates by setting bits or doing preparatory work for e.g. manual update display.

This hook is called before any plane commit functions are called.

Note that the power state of the display pipe when this function is called depends upon the exact helpers and calling sequence the driver has picked. See [`drm_atomic_helper_commit_planes\(\)`](#) for a discussion of the tradeoffs and variants of plane commit helpers.

This callback is used by the atomic modeset helpers, but it is optional.

`atomic_flush`

Drivers should finalize an atomic update of multiple planes on a CRTC in this hook. Depending upon hardware this might include checking that vblank evasion was successful, unblocking updates by setting bits or setting the GO bit to flush out all updates.

Simple hardware or hardware with special requirements can commit and flush out all updates for all planes from this hook and forgo all the other commit hooks for plane updates.

This hook is called after any plane commit functions are called.

Note that the power state of the display pipe when this function is called depends upon the exact helpers and calling sequence the driver has picked. See [`drm_atomic_helper_commit_planes\(\)`](#) for a discussion of the tradeoffs and variants of plane commit helpers.

This callback is used by the atomic modeset helpers, but it is optional.

`atomic_enable`

This callback should be used to enable the CRTC. With the atomic drivers it is called before all encoders connected to this CRTC are enabled through the encoder's own [`drm_encoder_helper_funcs.enable`](#) hook. If that sequence is too simple drivers can just add their own hooks and call it from this CRTC callback here by looping over all encoders connected to it using `for_each_encoder_on_crtc()`.

This hook is used only by atomic helpers, for symmetry with **`atomic_disable`**. Atomic drivers don't need to implement it if there's no need to enable anything at the CRTC level. To ensure that runtime PM handling (using either DPMS or the new "ACTIVE" property) works **`atomic_enable`** must be the inverse of **`atomic_disable`** for atomic drivers.

This function is optional.

`atomic_disable`

This callback should be used to disable the CRTC. With the atomic drivers it is called after all encoders connected to this CRTC have been shut off already using their own [`drm_encoder_helper_funcs.disable`](#) hook. If that sequence is too simple drivers can just

add their own hooks and call it from this CRTC callback here by looping over all encoders connected to it using `for_each_encoder_on_crtc()`.

This hook is used only by atomic helpers. Atomic drivers don't need to implement it if there's no need to disable anything at the CRTC level.

This function is optional.

get_scanout_position

Called by vblank timestamping code.

Returns the current display scanout position from a CRTC and an optional accurate `ktime_get()` timestamp of when the position was measured. Note that this is a helper callback which is only used if a driver uses `drm_crtc_vblank_helper_get_vblank_timestamp()` for the **`drm_crtc_funcs.get_vblank_timestamp`** callback.

Parameters:

crtc:

The CRTC.

in_vblank_irq:

True when called from `drm_crtc_handle_vblank()`. Some drivers need to apply some workarounds for gpu-specific vblank irq quirks if the flag is set.

vpos:

Target location for current vertical scanout position.

hpos:

Target location for current horizontal scanout position.

stime:

Target location for timestamp taken immediately before scanout position query. Can be NULL to skip timestamp.

etime:

Target location for timestamp taken immediately after scanout position query. Can be NULL to skip timestamp.

mode:

Current display timings.

Returns vpos as a positive number while in active scanout area. Returns vpos as a negative number inside vblank, counting the number of scanlines to go until end of vblank, e.g., -1 means "one scanline until start of active scanout / end of vblank."

Returns:

True on success, false if a reliable scanout position counter could not be read out.

Description

These hooks are used by the legacy CRTC helpers and the new atomic modesetting helpers.

void **drm_crtc_helper_add**(struct *drm_crtc* *crtc, const struct *drm_crtc_helper_funcs* *funcs)
sets the helper vtable for a crtc

Parameters

struct drm_crtc *crtc
DRM CRTC

const struct drm_crtc_helper_funcs *funcs
helper vtable to set for **crtc**

struct **drm_encoder_helper_funcs**
helper operations for encoders

Definition:

```
struct drm_encoder_helper_funcs {  
    void (*dpms)(struct drm_encoder *encoder, int mode);  
    enum drm_mode_status (*mode_valid)(struct drm_encoder *crtc, const struct  
→ drm_display_mode *mode);  
    bool (*mode_fixup)(struct drm_encoder *encoder, const struct drm_display_  
→ mode *mode, struct drm_display_mode *adjusted_mode);  
    void (*prepare)(struct drm_encoder *encoder);  
    void (*commit)(struct drm_encoder *encoder);  
    void (*mode_set)(struct drm_encoder *encoder, struct drm_display_mode *mode,  
→ struct drm_display_mode *adjusted_mode);  
    void (*atomic_mode_set)(struct drm_encoder *encoder, struct drm_crtc_state_  
→ *crtc_state, struct drm_connector_state *conn_state);  
    enum drm_connector_status (*detect)(struct drm_encoder *encoder, struct_  
→ drm_connector *connector);  
    void (*atomic_disable)(struct drm_encoder *encoder, struct drm_atomic_  
→ state *state);  
    void (*atomic_enable)(struct drm_encoder *encoder, struct drm_atomic_state_  
→ *state);  
    void (*disable)(struct drm_encoder *encoder);  
    void (*enable)(struct drm_encoder *encoder);  
    int (*atomic_check)(struct drm_encoder *encoder, struct drm_crtc_state_  
→ *crtc_state, struct drm_connector_state *conn_state);  
};
```

Members

dpms

Callback to control power levels on the encoder. If the mode passed in is unsupported, the provider must use the next lowest power level. This is used by the legacy encoder helpers to implement DPMS functionality in [drm_helper_connector_dpms\(\)](#).

This callback is also used to disable an encoder by calling it with `DRM_MODE_DPMS_OFF` if the **disable** hook isn't used.

This callback is used by the legacy CRTC helpers. Atomic helpers also support using this hook for enabling and disabling an encoder to facilitate transitions to atomic, but it is deprecated. Instead **enable** and **disable** should be used.

mode_valid

This callback is used to check if a specific mode is valid in this encoder. This should be implemented if the encoder has some sort of restriction in the modes it can display. For example, a given encoder may be responsible to set a clock value. If the clock can not produce all the values for the available modes then this callback can be used to restrict the number of modes to only the ones that can be displayed.

This hook is used by the probe helpers to filter the mode list in `drm_helper_probe_single_connector_modes()`, and it is used by the atomic helpers to validate modes supplied by userspace in `drm_atomic_helper_check_modeset()`.

This function is optional.

NOTE:

Since this function is both called from the check phase of an atomic commit, and the mode validation in the probe paths it is not allowed to look at anything else but the passed-in mode, and validate it against configuration-invariant hardware constraints. Any further limits which depend upon the configuration can only be checked in **mode_fixup** or **atomic_check**.

RETURNS:

`drm_mode_status` Enum

mode_fixup

This callback is used to validate and adjust a mode. The parameter `mode` is the display mode that should be fed to the next element in the display chain, either the final `drm_connector` or a `drm_bridge`. The parameter `adjusted_mode` is the input mode the encoder requires. It can be modified by this callback and does not need to match `mode`. See also `drm_crtc_state.adjusted_mode` for more details.

This function is used by both legacy CRTC helpers and atomic helpers. This hook is optional.

NOTE:

This function is called in the check phase of atomic modesets, which can be aborted for any reason (including on userspace's request to just check whether a configuration would be possible). Atomic drivers **MUST NOT** touch any persistent state (hardware or software) or data structures except the passed in `adjusted_mode` parameter.

This is in contrast to the legacy CRTC helpers where this was allowed.

Atomic drivers which need to inspect and adjust more state should instead use the **atomic_check** callback. If **atomic_check** is used, this hook isn't called since **atomic_check** allows a strict superset of the functionality of **mode_fixup**.

Also beware that userspace can request its own custom modes, neither core nor helpers filter modes to the list of probe modes reported by the GETCONNECTOR IOCTL and stored in `drm_connector.modes`. To ensure that modes are filtered consistently put any encoder constraints and limits checks into **mode_valid**.

RETURNS:

True if an acceptable configuration is possible, false if the modeset operation should be rejected.

prepare

This callback should prepare the encoder for a subsequent modeset, which in practice means the driver should disable the encoder if it is running. Most drivers ended up implementing this by calling their **dpms** hook with `DRM_MODE_DPMS_OFF`.

This callback is used by the legacy CRTC helpers. Atomic helpers also support using this hook for disabling an encoder to facilitate transitions to atomic, but it is deprecated. Instead **disable** should be used.

commit

This callback should commit the new mode on the encoder after a modeset, which in practice means the driver should enable the encoder. Most drivers ended up implementing this by calling their **dpms** hook with `DRM_MODE_DPMS_ON`.

This callback is used by the legacy CRTC helpers. Atomic helpers also support using this hook for enabling an encoder to facilitate transitions to atomic, but it is deprecated. Instead **enable** should be used.

mode_set

This callback is used to update the display mode of an encoder.

Note that the display pipe is completely off when this function is called. Drivers which need hardware to be running before they program the new display mode (because they implement runtime PM) should not use this hook, because the helper library calls it only once and not every time the display pipeline is suspend using either DPMS or the new “ACTIVE” property. Such drivers should instead move all their encoder setup into the **enable** callback.

This callback is used both by the legacy CRTC helpers and the atomic modeset helpers. It is optional in the atomic helpers.

NOTE:

If the driver uses the atomic modeset helpers and needs to inspect the connector state or connector display info during mode setting, **atomic_mode_set** can be used instead.

atomic_mode_set

This callback is used to update the display mode of an encoder.

Note that the display pipe is completely off when this function is called. Drivers which need hardware to be running before they program the new display mode (because they implement runtime PM) should not use this hook, because the helper library calls it only once and not every time the display pipeline is suspended using either DPMS or the new “ACTIVE” property. Such drivers should instead move all their encoder setup into the **enable** callback.

This callback is used by the atomic modeset helpers in place of the **mode_set** callback, if set by the driver. It is optional and should be used instead of **mode_set** if the driver needs to inspect the connector state or display info, since there is no direct way to go from the encoder to the current connector.

detect

This callback can be used by drivers who want to do detection on the encoder object instead of in connector functions.

It is not used by any helper and therefore has purely driver-specific semantics. New drivers shouldn't use this and instead just implement their own private callbacks.

FIXME:

This should just be converted into a pile of driver vfuncs. Currently radeon, amdgpu and nouveau are using it.

atomic_disable

This callback should be used to disable the encoder. With the atomic drivers it is called before this encoder's CRTC has been shut off using their own [drm_crtc_helper_funcs.atomic_disable](#) hook. If that sequence is too simple drivers can just add their own driver

private encoder hooks and call them from CRTC's callback by looping over all encoders connected to it using `for_each_encoder_on_crtc()`.

This callback is a variant of **disable** that provides the atomic state to the driver. If **atomic_disable** is implemented, **disable** is not called by the helpers.

This hook is only used by atomic helpers. Atomic drivers don't need to implement it if there's no need to disable anything at the encoder level. To ensure that runtime PM handling (using either DPMS or the new "ACTIVE" property) works **atomic_disable** must be the inverse of **atomic_enable**.

atomic_enable

This callback should be used to enable the encoder. It is called after this encoder's CRTC has been enabled using their own `drm_crtc_helper_funcs.atomic_enable` hook. If that sequence is too simple drivers can just add their own driver private encoder hooks and call them from CRTC's callback by looping over all encoders connected to it using `for_each_encoder_on_crtc()`.

This callback is a variant of **enable** that provides the atomic state to the driver. If **atomic_enable** is implemented, **enable** is not called by the helpers.

This hook is only used by atomic helpers, it is the opposite of **atomic_disable**. Atomic drivers don't need to implement it if there's no need to enable anything at the encoder level. To ensure that runtime PM handling works **atomic_enable** must be the inverse of **atomic_disable**.

disable

This callback should be used to disable the encoder. With the atomic drivers it is called before this encoder's CRTC has been shut off using their own `drm_crtc_helper_funcs.disable` hook. If that sequence is too simple drivers can just add their own driver private encoder hooks and call them from CRTC's callback by looping over all encoders connected to it using `for_each_encoder_on_crtc()`.

This hook is used both by legacy CRTC helpers and atomic helpers. Atomic drivers don't need to implement it if there's no need to disable anything at the encoder level. To ensure that runtime PM handling (using either DPMS or the new "ACTIVE" property) works **disable** must be the inverse of **enable** for atomic drivers.

For atomic drivers also consider **atomic_disable** and save yourself from having to read the NOTE below!

NOTE:

With legacy CRTC helpers there's a big semantic difference between **disable** and other hooks (like **prepare** or **dpms**) used to shut down a encoder: **disable** is only called when also logically disabling the display pipeline and needs to release any resources acquired in **mode_set** (like shared PLLs, or again release pinned framebuffers).

Therefore **disable** must be the inverse of **mode_set** plus **commit** for drivers still using legacy CRTC helpers, which is different from the rules under atomic.

enable

This callback should be used to enable the encoder. With the atomic drivers it is called after this encoder's CRTC has been enabled using their own `drm_crtc_helper_funcs.enable` hook. If that sequence is too simple drivers can just add their own driver private encoder hooks and call them from CRTC's callback by looping over all encoders connected to it using `for_each_encoder_on_crtc()`.

This hook is only used by atomic helpers, it is the opposite of **disable**. Atomic drivers don't need to implement it if there's no need to enable anything at the encoder level. To ensure that runtime PM handling (using either DPMS or the new "ACTIVE" property) works **enable** must be the inverse of **disable** for atomic drivers.

atomic_check

This callback is used to validate encoder state for atomic drivers. Since the encoder is the object connecting the CRTC and connector it gets passed both states, to be able to validate interactions and update the CRTC to match what the encoder needs for the requested connector.

Since this provides a strict superset of the functionality of **mode_fixup** (the requested and adjusted modes are both available through the passed in *struct drm_crtc_state*) **mode_fixup** is not called when **atomic_check** is implemented.

This function is used by the atomic helpers, but it is optional.

NOTE:

This function is called in the check phase of an atomic update. The driver is not allowed to change anything outside of the free-standing state objects passed-in or assembled in the overall *drm_atomic_state* update tracking structure.

Also beware that userspace can request its own custom modes, neither core nor helpers filter modes to the list of probe modes reported by the GETCONNECTOR IOCTL and stored in *drm_connector.modes*. To ensure that modes are filtered consistently put any encoder constraints and limits checks into **mode_valid**.

RETURNS:

0 on success, -EINVAL if the state or the transition can't be supported, -ENOMEM on memory allocation failure and -EDEADLK if an attempt to obtain another state object ran into a *drm_modeset_lock* deadlock.

Description

These hooks are used by the legacy CRTC helpers and the new atomic modesetting helpers.

```
void drm_encoder_helper_add(struct drm_encoder *encoder, const struct  
                           drm_encoder_helper_funcs *funcs)
```

sets the helper vtable for an encoder

Parameters

```
struct drm_encoder *encoder
```

DRM encoder

```
const struct drm_encoder_helper_funcs *funcs
```

helper vtable to set for **encoder**

```
struct drm_connector_helper_funcs
```

helper operations for connectors

Definition:

```
struct drm_connector_helper_funcs {  
    int (*get_modes)(struct drm_connector *connector);  
    int (*detect_ctx)(struct drm_connector *connector, struct drm_modeset_  
→ acquire_ctx *ctx, bool force);
```

```

enum drm_mode_status (*mode_valid)(struct drm_connector *connector, struct
↪drm_display_mode *mode);
int (*mode_valid_ctx)(struct drm_connector *connector, struct drm_display_
↪mode *mode, struct drm_modeset_acquire_ctx *ctx, enum drm_mode_status
↪*status);
struct drm_encoder *(*best_encoder)(struct drm_connector *connector);
struct drm_encoder *(*atomic_best_encoder)(struct drm_connector *connector,
↪ struct drm_atomic_state *state);
int (*atomic_check)(struct drm_connector *connector, struct drm_atomic_
↪state *state);
void (*atomic_commit)(struct drm_connector *connector, struct drm_atomic_
↪state *state);
int (*prepare_writeback_job)(struct drm_writeback_connector *connector,
↪ struct drm_writeback_job *job);
void (*cleanup_writeback_job)(struct drm_writeback_connector *connector,
↪ struct drm_writeback_job *job);
void (*enable_hpd)(struct drm_connector *connector);
void (*disable_hpd)(struct drm_connector *connector);
};

```

Members

get_modes

This function should fill in all modes currently valid for the sink into the *drm_connector.probed_modes* list. It should also update the EDID property by calling *drm_connector_update_edid_property()*.

The usual way to implement this is to cache the EDID retrieved in the probe callback somewhere in the driver-private connector structure. In this function drivers then parse the modes in the EDID and add them by calling *drm_add_edid_modes()*. But connectors that drive a fixed panel can also manually add specific modes using *drm_mode_probed_add()*. Drivers which manually add modes should also make sure that the *drm_connector.display_info*, *drm_connector.width_mm* and *drm_connector.height_mm* fields are filled in.

Note that the caller function will automatically add standard VESA DMT modes up to 1024x768 if the *.get_modes()* helper operation returns no mode and if the connector status is *connector_status_connected* or *connector_status_unknown*. There is no need to call *drm_add_modes_noedid()* manually in that case.

Virtual drivers that just want some standard VESA mode with a given resolution can call *drm_add_modes_noedid()*, and mark the preferred one using *drm_set_preferred_mode()*.

This function is only called after the **detect** hook has indicated that a sink is connected and when the EDID isn't overridden through sysfs or the kernel commandline.

This callback is used by the probe helpers in e.g. *drm_helper_probe_single_connector_modes()*

To avoid races with concurrent connector state updates, the helper libraries always call this with the *drm_mode_config.connection_mutex* held. Because of this it's safe to inspect *drm_connector->state*.

RETURNS:

The number of modes added by calling *drm_mode_probed_add()*. Return 0 on failures (no

modes) instead of negative error codes.

detect_ctx

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

This callback is optional, if not implemented the connector will be considered as always being attached.

This is the atomic version of *drm_connector_funcs.detect*.

To avoid races against concurrent connector state updates, the helper libraries always call this with *ctx* set to a valid context, and *drm_mode_config.connection_mutex* will always be locked with the *ctx* parameter set to this *ctx*. This allows taking additional locks as required.

RETURNS:

drm_connector_status indicating the connector's status, or the error code returned by *drm_modeset_lock()*, -EDEADLK.

mode_valid

Callback to validate a mode for a connector, irrespective of the specific display configuration.

This callback is used by the probe helpers to filter the mode list (which is usually derived from the EDID data block from the sink). See e.g. *drm_helper_probe_single_connector_modes()*.

This function is optional.

NOTE:

This only filters the mode list supplied to userspace in the GETCONNECTOR IOCTL. Compared to *drm_encoder_helper_funcs.mode_valid*, *drm_crtc_helper_funcs.mode_valid* and *drm_bridge_funcs.mode_valid*, which are also called by the atomic helpers from *drm_atomic_helper_check_modeset()*. This allows userspace to force and ignore sink constraint (like the pixel clock limits in the screen's EDID), which is useful for e.g. testing, or working around a broken EDID. Any source hardware constraint (which always need to be enforced) therefore should be checked in one of the above callbacks, and not this one here.

To avoid races with concurrent connector state updates, the helper libraries always call this with the *drm_mode_config.connection_mutex* held. Because of this it's safe to inspect *drm_connector->state*.

RETURNS:

Either *drm_mode_status.MODE_OK* or one of the failure reasons in *enum drm_mode_status*.

mode_valid_ctx

Callback to validate a mode for a connector, irrespective of the specific display configuration.

This callback is used by the probe helpers to filter the mode list (which is usually derived from the EDID data block from the sink). See e.g. *drm_helper_probe_single_connector_modes()*.

This function is optional, and is the atomic version of *drm_connector_helper_funcs.mode_valid*.

To allow for accessing the atomic state of modesetting objects, the helper libraries always call this with *ctx* set to a valid context, and *drm_mode_config.connection_mutex* will always be locked with the *ctx* parameter set to **ctx**. This allows for taking additional locks as required.

Even though additional locks may be acquired, this callback is still expected not to take any constraints into account which would be influenced by the currently set display state - such constraints should be handled in the driver's atomic check. For example, if a connector shares display bandwidth with other connectors then it would be ok to validate the minimum bandwidth requirement of a mode against the maximum possible bandwidth of the connector. But it wouldn't be ok to take the current bandwidth usage of other connectors into account, as this would change depending on the display state.

Returns: 0 if *drm_connector_helper_funcs.mode_valid_ctx* succeeded and wrote the *enum drm_mode_status* value to **status**, or a negative error code otherwise.

best_encoder

This function should select the best encoder for the given connector.

This function is used by both the atomic helpers (in the *drm_atomic_helper_check_modeset()* function) and in the legacy CRTC helpers.

NOTE:

In atomic drivers this function is called in the check phase of an atomic update. The driver is not allowed to change or inspect anything outside of arguments passed-in. Atomic drivers which need to inspect dynamic configuration state should instead use **atomic_best_encoder**.

You can leave this function to NULL if the connector is only attached to a single encoder. In this case, the core will call *drm_connector_get_single_encoder()* for you.

RETURNS:

Encoder that should be used for the given connector and connector state, or NULL if no suitable encoder exists. Note that the helpers will ensure that encoders aren't used twice, drivers should not check for this.

atomic_best_encoder

This is the atomic version of **best_encoder** for atomic drivers which need to select the best encoder depending upon the desired configuration and can't select it statically.

This function is used by *drm_atomic_helper_check_modeset()*. If it is not implemented, the core will fallback to **best_encoder** (or *drm_connector_get_single_encoder()* if **best_encoder** is NULL).

NOTE:

This function is called in the check phase of an atomic update. The driver is not allowed to change anything outside of the *drm_atomic_state* update tracking structure passed in.

RETURNS:

Encoder that should be used for the given connector and connector state, or NULL if no suitable encoder exists. Note that the helpers will ensure that encoders aren't used twice, drivers should not check for this.

atomic_check

This hook is used to validate connector state. This function is called from [drm_atomic_helper_check_modeset](#), and is called when a connector property is set, or a modeset on the crtc is forced.

Because [drm_atomic_helper_check_modeset](#) may be called multiple times, this function should handle being called multiple times as well.

This function is also allowed to inspect any other object's state and can add more state objects to the atomic commit if needed. Care must be taken though to ensure that state check and compute functions for these added states are all called, and derived state in other objects all updated. Again the recommendation is to just call check helpers until a maximal configuration is reached.

NOTE:

This function is called in the check phase of an atomic update. The driver is not allowed to change anything outside of the free-standing state objects passed-in or assembled in the overall [drm_atomic_state](#) update tracking structure.

RETURNS:

0 on success, -EINVAL if the state or the transition can't be supported, -ENOMEM on memory allocation failure and -EDEADLK if an attempt to obtain another state object ran into a [drm_modeset_lock](#) deadlock.

atomic_commit

This hook is to be used by drivers implementing writeback connectors that need a point when to commit the writeback job to the hardware. The writeback_job to commit is available in the new connector state, in [drm_connector_state.writeback_job](#).

This hook is optional.

This callback is used by the atomic modeset helpers.

prepare_writeback_job

As writeback jobs contain a framebuffer, drivers may need to prepare and clean them up the same way they can prepare and clean up framebuffers for planes. This optional connector operation is used to support the preparation of writeback jobs. The job prepare operation is called from [drm_atomic_helper_prepare_planes\(\)](#) for struct [drm_writeback_connector](#) connectors only.

This operation is optional.

This callback is used by the atomic modeset helpers.

cleanup_writeback_job

This optional connector operation is used to support the cleanup of writeback jobs. The job cleanup operation is called from the existing [drm_writeback_cleanup_job\(\)](#) function, invoked both when destroying the job as part of an aborted commit, or when the job completes.

This operation is optional.

This callback is used by the atomic modeset helpers.

enable_hpd

Enable hot-plug detection for the connector.

This operation is optional.

This callback is used by the `drm_kms_helper_poll_enable()` helpers.

disable_hpd

Disable hot-plug detection for the connector.

This operation is optional.

This callback is used by the `drm_kms_helper_poll_disable()` helpers.

Description

These functions are used by the atomic and legacy modeset helpers and by the probe helpers.

`void drm_connector_helper_add(struct drm_connector *connector, const struct drm_connector_helper_funcs *funcs)`

sets the helper vtable for a connector

Parameters

`struct drm_connector *connector`

DRM connector

`const struct drm_connector_helper_funcs *funcs`

helper vtable to set for **connector**

`struct drm_plane_helper_funcs`

helper operations for planes

Definition:

```
struct drm_plane_helper_funcs {
    int (*prepare_fb)(struct drm_plane *plane, struct drm_plane_state *new_
↪state);
    void (*cleanup_fb)(struct drm_plane *plane, struct drm_plane_state *old_
↪state);
    int (*begin_fb_access)(struct drm_plane *plane, struct drm_plane_state_
↪*new_plane_state);
    void (*end_fb_access)(struct drm_plane *plane, struct drm_plane_state *new_
↪plane_state);
    int (*atomic_check)(struct drm_plane *plane, struct drm_atomic_state_
↪*state);
    void (*atomic_update)(struct drm_plane *plane, struct drm_atomic_state_
↪*state);
    void (*atomic_enable)(struct drm_plane *plane, struct drm_atomic_state_
↪*state);
    void (*atomic_disable)(struct drm_plane *plane, struct drm_atomic_state_
↪*state);
    int (*atomic_async_check)(struct drm_plane *plane, struct drm_atomic_state_
↪*state);
    void (*atomic_async_update)(struct drm_plane *plane, struct drm_atomic_
↪state *state);
};
```

Members

prepare_fb

This hook is to prepare a framebuffer for scanout by e.g. pinning its backing storage or

relocating it into a contiguous block of VRAM. Other possible preparatory work includes flushing caches.

This function must not block for outstanding rendering, since it is called in the context of the atomic IOCTL even for async commits to be able to return any errors to userspace. Instead the recommended way is to fill out the *drm_plane_state.fence* of the passed-in *drm_plane_state*. If the driver doesn't support native fences then equivalent functionality should be implemented through private members in the plane structure.

For GEM drivers who neither have a **prepare_fb** nor **cleanup_fb** hook set *drm_gem_plane_helper_prepare_fb()* is called automatically to implement this. Other drivers which need additional plane processing can call *drm_gem_plane_helper_prepare_fb()* from their **prepare_fb** hook.

The resources acquired in **prepare_fb** persist after the end of the atomic commit. Resources that can be release at the commit's end should be acquired in **begin_fb_access** and released in **end_fb_access**. For example, a GEM buffer's pin operation belongs into **prepare_fb** to keep the buffer pinned after the commit. But a vmap operation for shadow-plane helpers belongs into **begin_fb_access**, so that atomic helpers remove the mapping at the end of the commit.

The helpers will call **cleanup_fb** with matching arguments for every successful call to this hook.

This callback is used by the atomic modeset helpers, but it is optional. See **begin_fb_access** for preparing per-commit resources.

RETURNS:

0 on success or one of the following negative error codes allowed by the *drm_mode_config_funcs.atomic_commit* vfunc. When using helpers this callback is the only one which can fail an atomic commit, everything else must complete successfully.

cleanup_fb

This hook is called to clean up any resources allocated for the given framebuffer and plane configuration in **prepare_fb**.

This callback is used by the atomic modeset helpers, but it is optional.

begin_fb_access

This hook prepares the plane for access during an atomic commit. In contrast to **prepare_fb**, resources acquired in **begin_fb_access**, are released at the end of the atomic commit in **end_fb_access**.

For example, with shadow-plane helpers, the GEM buffer's vmap operation belongs into **begin_fb_access**, so that the buffer's memory will be unmapped at the end of the commit in **end_fb_access**. But a GEM buffer's pin operation belongs into **prepare_fb** to keep the buffer pinned after the commit.

The callback is used by the atomic modeset helpers, but it is optional. See **end_fb_cleanup** for undoing the effects of **begin_fb_access** and **prepare_fb** for acquiring resources until the next pageflip.

Returns: 0 on success, or a negative errno code otherwise.

end_fb_access

This hook cleans up resources allocated by **begin_fb_access**. It is called at the end of a commit for the new plane state.

atomic_check

Drivers should check plane specific constraints in this hook.

When using `drm_atomic_helper_check_planes()` plane's **atomic_check** hooks are called before the ones for CRTC's, which allows drivers to request shared resources that the CRTC controls here. For more complicated dependencies the driver can call the provided check helpers multiple times until the computed state has a final configuration and everything has been checked.

This function is also allowed to inspect any other object's state and can add more state objects to the atomic commit if needed. Care must be taken though to ensure that state check and compute functions for these added states are all called, and derived state in other objects all updated. Again the recommendation is to just call check helpers until a maximal configuration is reached.

This callback is used by the atomic modeset helpers, but it is optional.

NOTE:

This function is called in the check phase of an atomic update. The driver is not allowed to change anything outside of the `drm_atomic_state` update tracking structure.

RETURNS:

0 on success, -EINVAL if the state or the transition can't be supported, -ENOMEM on memory allocation failure and -EDEADLK if an attempt to obtain another state object ran into a `drm_modeset_lock` deadlock.

atomic_update

Drivers should use this function to update the plane state. This hook is called in-between the `drm_crtc_helper_funcs.atomic_begin` and `drm_crtc_helper_funcs.atomic_flush` callbacks.

Note that the power state of the display pipe when this function is called depends upon the exact helpers and calling sequence the driver has picked. See `drm_atomic_helper_commit_planes()` for a discussion of the tradeoffs and variants of plane commit helpers.

This callback is used by the atomic modeset helpers, but it is optional.

atomic_enable

Drivers should use this function to unconditionally enable a plane. This hook is called in-between the `drm_crtc_helper_funcs.atomic_begin` and `drm_crtc_helper_funcs.atomic_flush` callbacks. It is called after **atomic_update**, which will be called for all enabled planes. Drivers that use **atomic_enable** should set up a plane in **atomic_update** and afterwards enable the plane in **atomic_enable**. If a plane needs to be enabled before installing the scanout buffer, drivers can still do so in **atomic_update**.

Note that the power state of the display pipe when this function is called depends upon the exact helpers and calling sequence the driver has picked. See `drm_atomic_helper_commit_planes()` for a discussion of the tradeoffs and variants of plane commit helpers.

This callback is used by the atomic modeset helpers, but it is optional. If implemented, **atomic_enable** should be the inverse of **atomic_disable**. Drivers that don't want to use either can still implement the complete plane update in **atomic_update**.

atomic_disable

Drivers should use this function to unconditionally disable a plane. This hook is called in-between the `drm_crtc_helper_funcs.atomic_begin` and `drm_crtc_helper_funcs.atomic_flush` callbacks. It is an alternative to **atomic_update**, which will be called for disabling planes, too, if the **atomic_disable** hook isn't implemented.

This hook is also useful to disable planes in preparation of a modeset, by calling `drm_atomic_helper_disable_planes_on_crtc()` from the `drm_crtc_helper_funcs.disable` hook.

Note that the power state of the display pipe when this function is called depends upon the exact helpers and calling sequence the driver has picked. See `drm_atomic_helper_commit_planes()` for a discussion of the tradeoffs and variants of plane commit helpers.

This callback is used by the atomic modeset helpers, but it is optional. It's intended to reverse the effects of **atomic_enable**.

atomic_async_check

Drivers should set this function pointer to check if the plane's atomic state can be updated in a async fashion. Here async means "not vblank synchronized".

This hook is called by `drm_atomic_async_check()` to establish if a given update can be committed asynchronously, that is, if it can jump ahead of the state currently queued for update.

RETURNS:

Return 0 on success and any error returned indicates that the update can not be applied in asynchronous manner.

atomic_async_update

Drivers should set this function pointer to perform asynchronous updates of planes, that is, jump ahead of the currently queued state and update the plane. Here async means "not vblank synchronized".

This hook is called by `drm_atomic_helper_async_commit()`.

An async update will happen on legacy cursor updates. An async update won't happen if there is an outstanding commit modifying the same plane.

When doing `async_update` drivers shouldn't replace the `drm_plane_state` but update the current one with the new plane configurations in the new `plane_state`.

Drivers should also swap the framebuffers between current plane state (`drm_plane.state`) and `new_state`. This is required since cleanup for async commits is performed on the new state, rather than old state like for traditional commits. Since we want to give up the reference on the current (old) fb instead of our brand new one, swap them in the driver during the async commit.

FIXME:

- It only works for single plane updates
- Async Pageflips are not supported yet
- Some hw might still scan out the old buffer until the next vblank, however we let go of the fb references as soon as we run this hook. For now drivers must implement

their own workers for deferring if needed, until a common solution is created.

Description

These functions are used by the atomic helpers.

void **drm_plane_helper_add**(struct *drm_plane* *plane, const struct *drm_plane_helper_funcs* *funcs)

sets the helper vtable for a plane

Parameters

struct drm_plane *plane
DRM plane

const struct drm_plane_helper_funcs *funcs
helper vtable to set for **plane**

struct drm_mode_config_helper_funcs
global modeset helper operations

Definition:

```
struct drm_mode_config_helper_funcs {
    void (*atomic_commit_tail)(struct drm_atomic_state *state);
    int (*atomic_commit_setup)(struct drm_atomic_state *state);
};
```

Members

atomic_commit_tail

This hook is used by the default `atomic_commit()` hook implemented in `drm_atomic_helper_commit()` together with the nonblocking commit helpers (see `drm_atomic_helper_setup_commit()` for a starting point) to implement blocking and nonblocking commits easily. It is not used by the atomic helpers

This function is called when the new atomic state has already been swapped into the various state pointers. The passed in state therefore contains copies of the old/previous state. This hook should commit the new state into hardware. Note that the helpers have already waited for preceeding atomic commits and fences, but drivers can add more waiting calls at the start of their implementation, e.g. to wait for driver-internal request for implicit syncing, before starting to commit the update to the hardware.

After the atomic update is committed to the hardware this hook needs to call `drm_atomic_helper_commit_hw_done()`. Then wait for the update to be executed by the hardware, for example using `drm_atomic_helper_wait_for_vblanks()` or `drm_atomic_helper_wait_for_flip_done()`, and then clean up the old framebuffers using `drm_atomic_helper_cleanup_planes()`.

When disabling a CRTC this hook `_must_` stall for the commit to complete. Vblank waits don't work on disabled CRTC, hence the core can't take care of this. And it also can't rely on the vblank event, since that can be signalled already when the screen shows black, which can happen much earlier than the last hardware access needed to shut off the display pipeline completely.

This hook is optional, the default implementation is `drm_atomic_helper_commit_tail()`.

atomic_commit_setup

This hook is used by the default `atomic_commit()` hook implemented in `drm_atomic_helper_commit()` together with the nonblocking helpers (see `drm_atomic_helper_setup_commit()`) to extend the DRM commit setup. It is not used by the atomic helpers.

This function is called at the end of `drm_atomic_helper_setup_commit()`, so once the commit has been properly setup across the generic DRM object states. It allows drivers to do some additional commit tracking that isn't related to a CRTC, plane or connector, tracked in a `drm_private_obj` structure.

Note that the documentation of `drm_private_obj` has more details on how one should implement this.

This hook is optional.

Description

These helper functions are used by the atomic helpers.

5.2 Atomic Modeset Helper Functions Reference

5.2.1 Overview

This helper library provides implementations of check and commit functions on top of the CRTC modeset helper callbacks and the plane helper callbacks. It also provides convenience implementations for the atomic state handling callbacks for drivers which don't need to subclass the drm core structures to add their own additional internal state.

This library also provides default implementations for the check callback in `drm_atomic_helper_check()` and for the commit callback with `drm_atomic_helper_commit()`. But the individual stages and callbacks are exposed to allow drivers to mix and match and e.g. use the plane helpers only together with a driver private modeset implementation.

This library also provides implementations for all the legacy driver interfaces on top of the atomic interface. See `drm_atomic_helper_set_config()`, `drm_atomic_helper_disable_plane()`, and the various functions to implement `set_property` callbacks. New drivers must not implement these functions themselves but must use the provided helpers.

The atomic helper uses the same function table structures as all other modesetting helpers. See the documentation for `struct drm_crtc_helper_funcs`, `struct drm_encoder_helper_funcs` and `struct drm_connector_helper_funcs`. It also shares the `struct drm_plane_helper_funcs` function table with the plane helpers.

5.2.2 Implementing Asynchronous Atomic Commit

Nonblocking atomic commits should use struct `drm_crtc_commit` to sequence different operations against each another. Locks, especially struct `drm_modeset_lock`, should not be held in worker threads or any other asynchronous context used to commit the hardware state.

`drm_atomic_helper_commit()` implements the recommended sequence for nonblocking commits, using `drm_atomic_helper_setup_commit()` internally:

1. Run `drm_atomic_helper_prepare_planes()`. Since this can fail and we need to propagate out of memory/VRAM errors to userspace, it must be called synchronously.
2. Synchronize with any outstanding nonblocking commit worker threads which might be affected by the new state update. This is handled by `drm_atomic_helper_setup_commit()`.

Asynchronous workers need to have sufficient parallelism to be able to run different atomic commits on different CRTC's in parallel. The simplest way to achieve this is by running them on the `system_unbound_wq` work queue. Note that drivers are not required to split up atomic commits and run an individual commit in parallel - userspace is supposed to do that if it cares. But it might be beneficial to do that for modesets, since those necessarily must be done as one global operation, and enabling or disabling a CRTC can take a long time. But even that is not required.

IMPORTANT: A `drm_atomic_state` update for multiple CRTC's is sequenced against all CRTC's therein. Therefore for atomic state updates which only flip planes the driver must not get the struct `drm_crtc_state` of unrelated CRTC's in its atomic check code: This would prevent committing of atomic updates to multiple CRTC's in parallel. In general, adding additional state structures should be avoided as much as possible, because this reduces parallelism in (non-blocking) commits, both due to locking and due to commit sequencing requirements.

3. The software state is updated synchronously with `drm_atomic_helper_swap_state()`. Doing this under the protection of all modeset locks means concurrent callers never see inconsistent state. Note that commit workers do not hold any locks; their access is only coordinated through ordering. If workers would access state only through the pointers in the free-standing state objects (currently not the case for any driver) then even multiple pending commits could be in-flight at the same time.

4. Schedule a work item to do all subsequent steps, using the split-out commit helpers: a) pre-plane commit b) plane commit c) post-plane commit and then cleaning up the framebuffers after the old framebuffer is no longer being displayed. The scheduled work should synchronize against other workers using the `drm_crtc_commit` infrastructure as needed. See `drm_atomic_helper_setup_commit()` for more details.

5.2.3 Helper Functions Reference

`drm_atomic_crtc_for_each_plane`

`drm_atomic_crtc_for_each_plane` (plane, crtc)

iterate over planes currently attached to CRTC

Parameters

plane

the loop cursor

crtc

the CRTC whose planes are iterated

Description

This iterates over the current state, useful (for example) when applying atomic state after it has been checked and swapped. To iterate over the planes which *will* be attached (more useful in code called from [*drm_mode_config_funcs.atomic_check*](#)) see [*drm_atomic_crtc_state_for_each_plane\(\)*](#).

drm_atomic_crtc_state_for_each_plane

`drm_atomic_crtc_state_for_each_plane (plane, crtc_state)`

iterate over attached planes in new state

Parameters

plane

the loop cursor

crtc_state

the incoming CRTC state

Description

Similar to `drm_crtc_for_each_plane()`, but iterates the planes that will be attached if the specified state is applied. Useful during for example in code called from [*drm_mode_config_funcs.atomic_check*](#) operations, to validate the incoming state.

drm_atomic_crtc_state_for_each_plane_state

`drm_atomic_crtc_state_for_each_plane_state (plane, plane_state, crtc_state)`

iterate over attached planes in new state

Parameters

plane

the loop cursor

plane_state

loop cursor for the plane's state, must be const

crtc_state

the incoming CRTC state

Description

Similar to `drm_crtc_for_each_plane()`, but iterates the planes that will be attached if the specified state is applied. Useful during for example in code called from [*drm_mode_config_funcs.atomic_check*](#) operations, to validate the incoming state.

Compared to just [*drm_atomic_crtc_state_for_each_plane\(\)*](#) this also fills in a const `plane_state`. This is useful when a driver just wants to peek at other active planes on this CRTC, but does not need to change it.

`bool drm_atomic_plane_enabling(struct drm_plane_state *old_plane_state, struct drm_plane_state *new_plane_state)`

check whether a plane is being enabled

Parameters

struct drm_plane_state *old_plane_state
old atomic plane state

struct drm_plane_state *new_plane_state
new atomic plane state

Description

Checks the atomic state of a plane to determine whether it's being enabled or not. This also WARNs if it detects an invalid state (both CRTC and FB need to either both be NULL or both be non-NULL).

Return

True if the plane is being enabled, false otherwise.

bool **drm_atomic_plane_disabling**(struct *drm_plane_state* *old_plane_state, struct *drm_plane_state* *new_plane_state)
check whether a plane is being disabled

Parameters

struct drm_plane_state *old_plane_state
old atomic plane state

struct drm_plane_state *new_plane_state
new atomic plane state

Description

Checks the atomic state of a plane to determine whether it's being disabled or not. This also WARNs if it detects an invalid state (both CRTC and FB need to either both be NULL or both be non-NULL).

Return

True if the plane is being disabled, false otherwise.

int **drm_atomic_helper_check_modeset**(struct *drm_device* *dev, struct *drm_atomic_state* *state)
validate state object for modeset changes

Parameters

struct drm_device *dev
DRM device

struct drm_atomic_state *state
the driver state object

Description

Check the state object to see if the requested state is physically possible. This does all the CRTC and connector related computations for an atomic update and adds any additional connectors needed for full modesets. It calls the various per-object callbacks in the follow order:

1. *drm_connector_helper_funcs.atomic_best_encoder* for determining the new encoder.
2. *drm_connector_helper_funcs.atomic_check* to validate the connector state.
3. If it's determined a modeset is needed then all connectors on the affected CRTC are added and *drm_connector_helper_funcs.atomic_check* is run on them.

4. *drm_encoder_helper_funcs.mode_valid*, *drm_bridge_funcs.mode_valid* and *drm_crtc_helper_funcs.mode_valid* are called on the affected components.
5. *drm_bridge_funcs.mode_fixup* is called on all encoder bridges.
6. *drm_encoder_helper_funcs.atomic_check* is called to validate any encoder state. This function is only called when the encoder will be part of a configured CRTC, it must not be used for implementing connector property validation. If this function is NULL, *drm_atomic_encoder_helper_funcs.mode_fixup* is called instead.
7. *drm_crtc_helper_funcs.mode_fixup* is called last, to fix up the mode with CRTC constraints.

drm_crtc_state.mode_changed is set when the input mode is changed. *drm_crtc_state.connectors_changed* is set when a connector is added or removed from the CRTC. *drm_crtc_state.active_changed* is set when *drm_crtc_state.active* changes, which is used for DPMS. *drm_crtc_state.no_vblank* is set from the result of *drm_dev_has_vblank()*. See also: *drm_atomic_crtc_needs_modeset()*

IMPORTANT:

Drivers which set *drm_crtc_state.mode_changed* (e.g. in their *drm_plane_helper_funcs.atomic_check* hooks if a plane update can't be done without a full modeset) *must* call this function after that change. It is permitted to call this function multiple times for the same update, e.g. when the *drm_crtc_helper_funcs.atomic_check* functions depend upon the adjusted dotclock for fifo space allocation and watermark computation.

Return

Zero for success or -errno

int **drm_atomic_helper_check_wb_encoder_state**(struct *drm_encoder* *encoder, struct *drm_connector_state* *conn_state)

Check writeback encoder state

Parameters

struct *drm_encoder* *encoder
encoder state to check

struct *drm_connector_state* *conn_state
connector state to check

Description

Checks if the writeback connector state is valid, and returns an error if it isn't.

Return

Zero for success or -errno

int **drm_atomic_helper_check_plane_state**(struct *drm_plane_state* *plane_state, const struct *drm_crtc_state* *crtc_state, int min_scale, int max_scale, bool can_position, bool can_update_disabled)

Check plane state for validity

Parameters

struct drm_plane_state *plane_state

plane state to check

const struct drm_crtc_state *crtc_state

CRTC state to check

int min_scale

minimum **src:dest** scaling factor in 16.16 fixed point

int max_scale

maximum **src:dest** scaling factor in 16.16 fixed point

bool can_position

is it legal to position the plane such that it doesn't cover the entire CRTC? This will generally only be false for primary planes.

bool can_update_disabled

can the plane be updated while the CRTC is disabled?

Description

Checks that a desired plane update is valid, and updates various bits of derived state (clipped coordinates etc.). Drivers that provide their own plane handling rather than helper-provided implementations may still wish to call this function to avoid duplication of error checking code.

Return

Zero if update appears valid, error code on failure

int **drm_atomic_helper_check_crtc_primary_plane**(struct *drm_crtc_state* *crtc_state)

Check CRTC state for primary plane

Parameters

struct drm_crtc_state *crtc_state

CRTC state to check

Description

Checks that a CRTC has at least one primary plane attached to it, which is a requirement on some hardware. Note that this only involves the CRTC side of the test. To test if the primary plane is visible or if it can be updated without the CRTC being enabled, use *drm_atomic_helper_check_plane_state()* in the plane's atomic check.

Return

0 if a primary plane is attached to the CRTC, or an error code otherwise

int **drm_atomic_helper_check_planes**(struct *drm_device* *dev, struct *drm_atomic_state* *state)

validate state object for planes changes

Parameters

struct drm_device *dev

DRM device

struct drm_atomic_state *state

the driver state object

Description

Check the state object to see if the requested state is physically possible. This does all the plane update related checks using by calling into the `drm_crtc_helper_funcs.atomic_check` and `drm_plane_helper_funcs.atomic_check` hooks provided by the driver.

It also sets `drm_crtc_state.planes_changed` to indicate that a CRTC has updated planes.

Return

Zero for success or -errno

```
int drm_atomic_helper_check(struct drm_device *dev, struct drm_atomic_state *state)
    validate state object
```

Parameters

```
struct drm_device *dev
    DRM device
```

```
struct drm_atomic_state *state
    the driver state object
```

Description

Check the state object to see if the requested state is physically possible. Only CRTCs and planes have check callbacks, so for any additional (global) checking that a driver needs it can simply wrap that around this function. Drivers without such needs can directly use this as their `drm_mode_config_funcs.atomic_check` callback.

This just wraps the two parts of the state checking for planes and modeset state in the default order: First it calls `drm_atomic_helper_check_modeset()` and then `drm_atomic_helper_check_planes()`. The assumption is that the `drm_plane_helper_funcs.atomic_check` and `drm_crtc_helper_funcs.atomic_check` functions depend upon an updated `adjusted_mode.clock` to e.g. properly compute watermarks.

Note that zpos normalization will add all enable planes to the state which might not desired for some drivers. For example enable/disable of a cursor plane which have fixed zpos value would trigger all other enabled planes to be forced to the state change.

Return

Zero for success or -errno

```
void drm_atomic_helper_update_legacy_modeset_state(struct drm_device *dev, struct
                                                    drm_atomic_state *old_state)
    update legacy modeset state
```

Parameters

```
struct drm_device *dev
    DRM device
```

```
struct drm_atomic_state *old_state
    atomic state object with old state structures
```

Description

This function updates all the various legacy modeset state pointers in connectors, encoders and CRTCs.

Drivers can use this for building their own atomic commit if they don't have a pure helper-based modeset implementation.

Since these updates are not synchronized with lockings, only code paths called from *drm_mode_config_helper_funcs.atomic_commit_tail* can look at the legacy state filled out by this helper. Defacto this means this helper and the legacy state pointers are only really useful for transitioning an existing driver to the atomic world.

void **drm_atomic_helper_calc_timestamping_constants**(struct *drm_atomic_state* *state)
update vblank timestamping constants

Parameters

struct drm_atomic_state *state
atomic state object

Description

Updates the timestamping constants used for precise vblank timestamps by calling *drm_calc_timestamping_constants()* for all enabled crtcs in **state**.

void **drm_atomic_helper_commit_modeset_disables**(struct *drm_device* *dev, struct *drm_atomic_state* *old_state)
modeset commit to disable outputs

Parameters

struct drm_device *dev
DRM device

struct drm_atomic_state *old_state
atomic state object with old state structures

Description

This function shuts down all the outputs that need to be shut down and prepares them (if required) with the new mode.

For compatibility with legacy CRTC helpers this should be called before *drm_atomic_helper_commit_planes()*, which is what the default commit function does. But drivers with different needs can group the modeset commits together and do the plane commits at the end. This is useful for drivers doing runtime PM since planes updates then only happen when the CRTC is actually enabled.

void **drm_atomic_helper_commit_modeset_enables**(struct *drm_device* *dev, struct *drm_atomic_state* *old_state)
modeset commit to enable outputs

Parameters

struct drm_device *dev
DRM device

struct drm_atomic_state *old_state
atomic state object with old state structures

Description

This function enables all the outputs with the new configuration which had to be turned off for the update.

For compatibility with legacy CRTC helpers this should be called after `drm_atomic_helper_commit_planes()`, which is what the default commit function does. But drivers with different needs can group the modeset commits together and do the plane commits at the end. This is useful for drivers doing runtime PM since planes updates then only happen when the CRTC is actually enabled.

int **drm_atomic_helper_wait_for_fences**(struct *drm_device* *dev, struct *drm_atomic_state* *state, bool pre_swap)

wait for fences stashed in plane state

Parameters

struct drm_device *dev

DRM device

struct drm_atomic_state *state

atomic state object with old state structures

bool pre_swap

If true, do an interruptible wait, and **state** is the new state. Otherwise **state** is the old state.

Description

For implicit sync, driver should fish the exclusive fence out from the incoming fb's and stash it in the `drm_plane_state`. This is called after `drm_atomic_helper_swap_state()` so it uses the current plane state (and just uses the atomic state to find the changed planes)

Note that **pre_swap** is needed since the point where we block for fences moves around depending upon whether an atomic commit is blocking or non-blocking. For non-blocking commit all waiting needs to happen after `drm_atomic_helper_swap_state()` is called, but for blocking commits we want to wait **before** we do anything that can't be easily rolled back. That is before we call `drm_atomic_helper_swap_state()`.

Returns zero if success or < 0 if `dma_fence_wait()` fails.

void **drm_atomic_helper_wait_for_vblanks**(struct *drm_device* *dev, struct *drm_atomic_state* *old_state)

wait for vblank on CRTCs

Parameters

struct drm_device *dev

DRM device

struct drm_atomic_state *old_state

atomic state object with old state structures

Description

Helper to, after atomic commit, wait for vblanks on all affected CRTCs (ie. before cleaning up old framebuffers using `drm_atomic_helper_cleanup_planes()`). It will only wait on CRTCs where the framebuffers have actually changed to optimize for the legacy cursor and plane update use-case.

Drivers using the nonblocking commit tracking support initialized by calling `drm_atomic_helper_setup_commit()` should look at `drm_atomic_helper_wait_for_flip_done()` as an alternative.

```
void drm_atomic_helper_wait_for_flip_done(struct drm_device *dev, struct  
                                           drm_atomic_state *old_state)
```

wait for all page flips to be done

Parameters

```
struct drm_device *dev
```

DRM device

```
struct drm_atomic_state *old_state
```

atomic state object with old state structures

Description

Helper to, after atomic commit, wait for page flips on all affected crtcs (ie. before cleaning up old framebuffers using *drm_atomic_helper_cleanup_planes()*). Compared to *drm_atomic_helper_wait_for_vblanks()* this waits for the completion on all CRTC's, assuming that cursors-only updates are signalling their completion immediately (or using a different path).

This requires that drivers use the nonblocking commit tracking support initialized using *drm_atomic_helper_setup_commit()*.

```
void drm_atomic_helper_commit_tail(struct drm_atomic_state *old_state)
```

commit atomic update to hardware

Parameters

```
struct drm_atomic_state *old_state
```

atomic state object with old state structures

Description

This is the default implementation for the *drm_mode_config_helper_funcs.atomic_commit_tail* hook, for drivers that do not support runtime_pm or do not need the CRTC to be enabled to perform a commit. Otherwise, see *drm_atomic_helper_commit_tail_rpm()*.

Note that the default ordering of how the various stages are called is to match the legacy modeset helper library closest.

```
void drm_atomic_helper_commit_tail_rpm(struct drm_atomic_state *old_state)
```

commit atomic update to hardware

Parameters

```
struct drm_atomic_state *old_state
```

new modeset state to be committed

Description

This is an alternative implementation for the *drm_mode_config_helper_funcs.atomic_commit_tail* hook, for drivers that support runtime_pm or need the CRTC to be enabled to perform a commit. Otherwise, one should use the default implementation *drm_atomic_helper_commit_tail()*.

```
int drm_atomic_helper_async_check(struct drm_device *dev, struct drm_atomic_state  
                                  *state)
```

check if state can be committed asynchronously

Parameters

struct drm_device *dev
DRM device

struct drm_atomic_state *state
the driver state object

Description

This helper will check if it is possible to commit the state asynchronously. Async commits are not supposed to swap the states like normal sync commits but just do in-place changes on the current state.

It will return 0 if the commit can happen in an asynchronous fashion or error if not. Note that error just mean it can't be committed asynchronously, if it fails the commit should be treated like a normal synchronous commit.

void **drm_atomic_helper_async_commit**(struct *drm_device* *dev, struct *drm_atomic_state* *state)

commit state asynchronously

Parameters

struct drm_device *dev
DRM device

struct drm_atomic_state *state
the driver state object

Description

This function commits a state asynchronously, i.e., not vblank synchronized. It should be used on a state only when `drm_atomic_async_check()` succeeds. Async commits are not supposed to swap the states like normal sync commits, but just do in-place changes on the current state.

TODO: Implement full swap instead of doing in-place changes.

int **drm_atomic_helper_commit**(struct *drm_device* *dev, struct *drm_atomic_state* *state, bool nonblock)

commit validated state object

Parameters

struct drm_device *dev
DRM device

struct drm_atomic_state *state
the driver state object

bool nonblock
whether nonblocking behavior is requested.

Description

This function commits a with *drm_atomic_helper_check()* pre-validated state object. This can still fail when e.g. the framebuffer reservation fails. This function implements nonblocking commits, using *drm_atomic_helper_setup_commit()* and related functions.

Committing the actual hardware state is done through the *drm_mode_config_helper_funcs.atomic_commit_tail* callback, or its default implementation *drm_atomic_helper_commit_tail()*.

Return

Zero for success or -errno.

int **drm_atomic_helper_setup_commit**(struct *drm_atomic_state* *state, bool nonblock)
setup possibly nonblocking commit

Parameters

struct **drm_atomic_state** *state
new modeset state to be committed

bool **nonblock**
whether nonblocking behavior is requested.

Description

This function prepares **state** to be used by the atomic helper's support for nonblocking commits. Drivers using the nonblocking commit infrastructure should always call this function from their *drm_mode_config_funcs.atomic_commit* hook.

Drivers that need to extend the commit setup to private objects can use the *drm_mode_config_helper_funcs.atomic_commit_setup* hook.

To be able to use this support drivers need to use a few more helper functions. *drm_atomic_helper_wait_for_dependencies()* must be called before actually committing the hardware state, and for nonblocking commits this call must be placed in the async worker. See also *drm_atomic_helper_swap_state()* and its stall parameter, for when a driver's commit hooks look at the *drm_crtc.state*, *drm_plane.state* or *drm_connector.state* pointer directly.

Completion of the hardware commit step must be signalled using *drm_atomic_helper_commit_hw_done()*. After this step the driver is not allowed to read or change any permanent software or hardware modeset state. The only exception is state protected by other means than *drm_modeset_lock* locks. Only the free standing **state** with pointers to the old state structures can be inspected, e.g. to clean up old buffers using *drm_atomic_helper_cleanup_planes()*.

At the very end, before cleaning up **state** drivers must call *drm_atomic_helper_commit_cleanup_done()*.

This is all implemented by in *drm_atomic_helper_commit()*, giving drivers a complete and easy-to-use default implementation of the atomic_commit() hook.

The tracking of asynchronously executed and still pending commits is done using the core structure *drm_crtc_commit*.

By default there's no need to clean up resources allocated by this function explicitly: *drm_atomic_state_default_clear()* will take care of that automatically.

0 on success. -EBUSY when userspace schedules nonblocking commits too fast, -ENOMEM on allocation failures and -EINTR when a signal is pending.

Return

void **drm_atomic_helper_wait_for_dependencies**(struct *drm_atomic_state* *old_state)
wait for required preceeding commits

Parameters

struct `drm_atomic_state` *old_state

atomic state object with old state structures

Description

This function waits for all preceeding commits that touch the same CRTC as **old_state** to both be committed to the hardware (as signalled by `drm_atomic_helper_commit_hw_done()`) and executed by the hardware (as signalled by calling `drm_crtc_send_vblank_event()` on the `drm_crtc_state.event`).

This is part of the atomic helper support for nonblocking commits, see `drm_atomic_helper_setup_commit()` for an overview.

void **drm_atomic_helper_fake_vblank**(struct `drm_atomic_state` *old_state)

fake VBLANK events if needed

Parameters

struct `drm_atomic_state` *old_state

atomic state object with old state structures

Description

This function walks all CRTCs and fakes VBLANK events on those with `drm_crtc_state.no_vblank` set to true and `drm_crtc_state.event` != NULL. The primary use of this function is writeback connectors working in oneshot mode and faking VBLANK events. In this case they only fake the VBLANK event when a job is queued, and any change to the pipeline that does not touch the connector is leading to timeouts when calling `drm_atomic_helper_wait_for_vblanks()` or `drm_atomic_helper_wait_for_flip_done()`. In addition to writeback connectors, this function can also fake VBLANK events for CRTCs without VBLANK interrupt.

This is part of the atomic helper support for nonblocking commits, see `drm_atomic_helper_setup_commit()` for an overview.

void **drm_atomic_helper_commit_hw_done**(struct `drm_atomic_state` *old_state)

setup possible nonblocking commit

Parameters

struct `drm_atomic_state` *old_state

atomic state object with old state structures

Description

This function is used to signal completion of the hardware commit step. After this step the driver is not allowed to read or change any permanent software or hardware modeset state. The only exception is state protected by other means than `drm_modeset_lock` locks.

Drivers should try to postpone any expensive or delayed cleanup work after this function is called.

This is part of the atomic helper support for nonblocking commits, see `drm_atomic_helper_setup_commit()` for an overview.

void **drm_atomic_helper_commit_cleanup_done**(struct `drm_atomic_state` *old_state)

signal completion of commit

Parameters

struct `drm_atomic_state` *old_state
atomic state object with old state structures

Description

This signals completion of the atomic update **old_state**, including any cleanup work. If used, it must be called right before calling [`drm_atomic_state_put\(\)`](#).

This is part of the atomic helper support for nonblocking commits, see [`drm_atomic_helper_setup_commit\(\)`](#) for an overview.

int **drm_atomic_helper_prepare_planes**(struct [`drm_device`](#) *dev, struct [`drm_atomic_state`](#) *state)
prepare plane resources before commit

Parameters

struct `drm_device` *dev
DRM device

struct `drm_atomic_state` *state
atomic state object with new state structures

Description

This function prepares plane state, specifically framebuffers, for the new configuration, by calling [`drm_plane_helper_funcs.prepare_fb`](#). If any failure is encountered this function will call [`drm_plane_helper_funcs.cleanup_fb`](#) on any already successfully prepared framebuffer.

Return

0 on success, negative error code on failure.

void **drm_atomic_helper_unprepare_planes**(struct [`drm_device`](#) *dev, struct [`drm_atomic_state`](#) *state)
release plane resources on aborts

Parameters

struct `drm_device` *dev
DRM device

struct `drm_atomic_state` *state
atomic state object with old state structures

Description

This function cleans up plane state, specifically framebuffers, from the atomic state. It undoes the effects of [`drm_atomic_helper_prepare_planes\(\)`](#) when aborting an atomic commit. For cleaning up after a successful commit use [`drm_atomic_helper_cleanup_planes\(\)`](#).

void **drm_atomic_helper_commit_planes**(struct [`drm_device`](#) *dev, struct [`drm_atomic_state`](#) *old_state, uint32_t flags)
commit plane state

Parameters

struct `drm_device` *dev
DRM device

struct drm_atomic_state *old_state

atomic state object with old state structures

uint32_t flags

flags for committing plane state

Description

This function commits the new plane state using the plane and atomic helper functions for planes and CRTC. It assumes that the atomic state has already been pushed into the relevant object state pointers, since this step can no longer fail.

It still requires the global state object **old_state** to know which planes and crtcs need to be updated though.

Note that this function does all plane updates across all CRTC. If the hardware can't support this approach look at [drm_atomic_helper_commit_planes_on_crtc\(\)](#) instead.

Plane parameters can be updated by applications while the associated CRTC is disabled. The DRM/KMS core will store the parameters in the plane state, which will be available to the driver when the CRTC is turned on. As a result most drivers don't need to be immediately notified of plane updates for a disabled CRTC.

Unless otherwise needed, drivers are advised to set the ACTIVE_ONLY flag in **flags** in order not to receive plane update notifications related to a disabled CRTC. This avoids the need to manually ignore plane updates in driver code when the driver and/or hardware can't or just don't need to deal with updates on disabled CRTC, for example when supporting runtime PM.

Drivers may set the NO_DISABLE_AFTER_MODESET flag in **flags** if the relevant display controllers require to disable a CRTC's planes when the CRTC is disabled. This function would skip the [drm_plane_helper_funcs.atomic_disable](#) call for a plane if the CRTC of the old plane state needs a modesetting operation. Of course, the drivers need to disable the planes in their CRTC disable callbacks since no one else would do that.

The [drm_atomic_helper_commit\(\)](#) default implementation doesn't set the ACTIVE_ONLY flag to most closely match the behaviour of the legacy helpers. This should not be copied blindly by drivers.

void drm_atomic_helper_commit_planes_on_crtc(struct [drm_crtc_state](#) *old_crtc_state)

commit plane state for a CRTC

Parameters

struct drm_crtc_state *old_crtc_state

atomic state object with the old CRTC state

Description

This function commits the new plane state using the plane and atomic helper functions for planes on the specific CRTC. It assumes that the atomic state has already been pushed into the relevant object state pointers, since this step can no longer fail.

This function is useful when plane updates should be done CRTC-by-CRTC instead of one global step like [drm_atomic_helper_commit_planes\(\)](#) does.

This function can only be safely used when planes are not allowed to move between different CRTC because this function doesn't handle inter-CRTC dependencies. Callers need to ensure that either no such dependencies exist, resolve them through ordering of commit calls or through some other means.

```
void drm_atomic_helper_disable_planes_on_crtc(struct drm_crtc_state *old_crtc_state,  
                                              bool atomic)
```

helper to disable CRTC's planes

Parameters

struct *drm_crtc_state* *old_crtc_state
atomic state object with the old CRTC state

bool atomic
if set, synchronize with CRTC's atomic_begin/flush hooks

Description

Disables all planes associated with the given CRTC. This can be used for instance in the CRTC helper atomic_disable callback to disable all planes.

If the atomic-parameter is set the function calls the CRTC's atomic_begin hook before and atomic_flush hook after disabling the planes.

It is a bug to call this function without having implemented the *drm_plane_helper_funcs.atomic_disable* plane hook.

```
void drm_atomic_helper_cleanup_planes(struct drm_device *dev, struct drm_atomic_state  
                                      *old_state)
```

cleanup plane resources after commit

Parameters

struct *drm_device* *dev
DRM device

struct *drm_atomic_state* *old_state
atomic state object with old state structures

Description

This function cleans up plane state, specifically framebuffers, from the old configuration. Hence the old configuration must be perserved in **old_state** to be able to call this function.

This function may not be called on the new state when the atomic update fails at any point after calling *drm_atomic_helper_prepare_planes()*. Use *drm_atomic_helper_unprepare_planes()* in this case.

```
int drm_atomic_helper_swap_state(struct drm_atomic_state *state, bool stall)  
store atomic state into current sw state
```

Parameters

struct *drm_atomic_state* *state
atomic state

bool stall
stall for preceding commits

Description

This function stores the atomic state into the current state pointers in all driver objects. It should be called after all failing steps have been done and succeeded, but before the actual hardware state is committed.

For cleanup and error recovery the current state for all changed objects will be swapped into **state**.

With that sequence it fits perfectly into the plane prepare/cleanup sequence:

1. Call `drm_atomic_helper_prepare_planes()` with the staged atomic state.
2. Do any other steps that might fail.
3. Put the staged state into the current state pointers with this function.
4. Actually commit the hardware state.
5. Call `drm_atomic_helper_cleanup_planes()` with **state**, which since step 3 contains the old state. Also do any other cleanup required with that state.

stall must be set when nonblocking commits for this driver directly access the `drm_plane.state`, `drm_crtc.state` or `drm_connector.state` pointer. With the current atomic helpers this is almost always the case, since the helpers don't pass the right state structures to the callbacks.

Returns 0 on success. Can return -ERESTARTSYS when **stall** is true and the waiting for the previous commits has been interrupted.

Return

```
int drm_atomic_helper_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, struct
                                   drm_modeset_acquire_ctx *ctx)
```

Helper for primary plane update using atomic

Parameters

struct drm_plane *plane
plane object to update

struct drm_crtc *crtc
owning CRTC of owning plane

struct drm_framebuffer *fb
framebuffer to flip onto plane

int crtc_x
x offset of primary plane on **crtc**

int crtc_y
y offset of primary plane on **crtc**

unsigned int crtc_w
width of primary plane rectangle on **crtc**

unsigned int crtc_h
height of primary plane rectangle on **crtc**

uint32_t src_x
x offset of **fb** for panning

uint32_t src_y
y offset of **fb** for panning

uint32_t src_w

width of source rectangle in **fb**

uint32_t src_h

height of source rectangle in **fb**

struct drm_modeset_acquire_ctx *ctx

lock acquire context

Description

Provides a default plane update handler using the atomic driver interface.

Return

Zero on success, error code on failure

int **drm_atomic_helper_disable_plane**(struct *drm_plane* *plane, struct *drm_modeset_acquire_ctx* *ctx)

Helper for primary plane disable using atomic

Parameters

struct drm_plane *plane

plane to disable

struct drm_modeset_acquire_ctx *ctx

lock acquire context

Description

Provides a default plane disable handler using the atomic driver interface.

Return

Zero on success, error code on failure

int **drm_atomic_helper_set_config**(struct *drm_mode_set* *set, struct *drm_modeset_acquire_ctx* *ctx)

set a new config from userspace

Parameters

struct drm_mode_set *set

mode set configuration

struct drm_modeset_acquire_ctx *ctx

lock acquisition context

Description

Provides a default CRTC set_config handler using the atomic driver interface.

NOTE

For backwards compatibility with old userspace this automatically resets the “link-status” property to GOOD, to force any link re-training. The SETCRTC ioctl does not define whether an update does need a full modeset or just a plane update, hence we’re allowed to do that. See also *drm_connector_set_link_status_property()*.

Return

Returns 0 on success, negative errno numbers on failure.


```
int drm_atomic_helper_disable_all(struct drm_device *dev, struct  
                                drm_modeset_acquire_ctx *ctx)
```

disable all currently active outputs

Parameters

```
struct drm_device *dev  
    DRM device
```

```
struct drm_modeset_acquire_ctx *ctx  
    lock acquisition context
```

Description

Loops through all connectors, finding those that aren't turned off and then turns them off by setting their DPMS mode to OFF and deactivating the CRTC that they are connected to.

This is used for example in suspend/resume to disable all currently active functions when suspending. If you just want to shut down everything at e.g. driver unload, look at [drm_atomic_helper_shutdown\(\)](#).

Note that if callers haven't already acquired all modeset locks this might return -EDEADLK, which must be handled by calling [drm_modeset_backoff\(\)](#).

See also: [drm_atomic_helper_suspend\(\)](#), [drm_atomic_helper_resume\(\)](#) and [drm_atomic_helper_shutdown\(\)](#).

Return

0 on success or a negative error code on failure.

```
void drm_atomic_helper_shutdown(struct drm_device *dev)  
    shutdown all CRTC
```

Parameters

```
struct drm_device *dev  
    DRM device
```

Description

This shuts down all CRTC, which is useful for driver unloading. Shutdown on suspend should instead be handled with [drm_atomic_helper_suspend\(\)](#), since that also takes a snapshot of the modeset state to be restored on resume.

This is just a convenience wrapper around [drm_atomic_helper_disable_all\(\)](#), and it is the atomic version of [drm_helper_force_disable_all\(\)](#).

```
struct drm_atomic_state *drm_atomic_helper_duplicate_state(struct drm_device *dev,  
                                                         struct  
                                                         drm_modeset_acquire_ctx  
                                                         *ctx)
```

duplicate an atomic state object

Parameters

```
struct drm_device *dev  
    DRM device
```

```
struct drm_modeset_acquire_ctx *ctx  
    lock acquisition context
```


Description

Makes a copy of the current atomic state by looping over all objects and duplicating their respective states. This is used for example by suspend/ resume support code to save the state prior to suspend such that it can be restored upon resume.

Note that this treats atomic state as persistent between save and restore. Drivers must make sure that this is possible and won't result in confusion or erroneous behaviour.

Note that if callers haven't already acquired all modeset locks this might return -EDEADLK, which must be handled by calling [drm_modeset_backoff\(\)](#).

See also: [drm_atomic_helper_suspend\(\)](#), [drm_atomic_helper_resume\(\)](#)

Return

A pointer to the copy of the atomic state object on success or an ERR_PTR()-encoded error code on failure.

```
struct drm\_atomic\_state *drm_atomic_helper_suspend(struct drm\_device *dev)  
    subsystem-level suspend helper
```

Parameters

```
struct drm_device *dev  
    DRM device
```

Description

Duplicates the current atomic state, disables all active outputs and then returns a pointer to the original atomic state to the caller. Drivers can pass this pointer to the [drm_atomic_helper_resume\(\)](#) helper upon resume to restore the output configuration that was active at the time the system entered suspend.

Note that it is potentially unsafe to use this. The atomic state object returned by this function is assumed to be persistent. Drivers must ensure that this holds true. Before calling this function, drivers must make sure to suspend fbdev emulation so that nothing can be using the device.

See also: [drm_atomic_helper_duplicate_state\(\)](#), [drm_atomic_helper_disable_all\(\)](#), [drm_atomic_helper_resume\(\)](#), [drm_atomic_helper_commit_duplicated_state\(\)](#)

Return

A pointer to a copy of the state before suspend on success or an ERR_PTR()- encoded error code on failure. Drivers should store the returned atomic state object and pass it to the [drm_atomic_helper_resume\(\)](#) helper upon resume.

```
int drm_atomic_helper_commit_duplicated_state(struct drm\_atomic\_state *state, struct  
                                             drm\_modeset\_acquire\_ctx *ctx)  
    commit duplicated state
```

Parameters

```
struct drm_atomic_state *state  
    duplicated atomic state to commit  
  
struct drm_modeset_acquire_ctx *ctx  
    pointer to acquire_ctx to use for commit.
```

Description

The state returned by `drm_atomic_helper_duplicate_state()` and `drm_atomic_helper_suspend()` is partially invalid, and needs to be fixed up before commit.

See also: `drm_atomic_helper_suspend()`

Return

0 on success or a negative error code on failure.

int **drm_atomic_helper_resume**(struct *drm_device* *dev, struct *drm_atomic_state* *state)
subsystem-level resume helper

Parameters

struct drm_device *dev
DRM device

struct drm_atomic_state *state
atomic state to resume to

Description

Calls `drm_mode_config_reset()` to synchronize hardware and software states, grabs all modeset locks and commits the atomic state object. This can be used in conjunction with the `drm_atomic_helper_suspend()` helper to implement suspend/resume for drivers that support atomic mode-setting.

See also: `drm_atomic_helper_suspend()`

Return

0 on success or a negative error code on failure.

int **drm_atomic_helper_page_flip**(struct *drm_crtc* *crtc, struct *drm_framebuffer* *fb, struct *drm_pending_vblank_event* *event, uint32_t flags, struct *drm_modeset_acquire_ctx* *ctx)
execute a legacy page flip

Parameters

struct drm_crtc *crtc
DRM CRTC

struct drm_framebuffer *fb
DRM framebuffer

struct drm_pending_vblank_event *event
optional DRM event to signal upon completion

uint32_t flags
flip flags for non-vblank sync'ed updates

struct drm_modeset_acquire_ctx *ctx
lock acquisition context

Description

Provides a default `drm_crtc_funcs.page_flip` implementation using the atomic driver interface.

See also: `drm_atomic_helper_page_flip_target()`

Return

Returns 0 on success, negative errno numbers on failure.

```
int drm_atomic_helper_page_flip_target(struct drm_crtc *crtc, struct drm_framebuffer
                                     *fb, struct drm_pending_vblank_event *event,
                                     uint32_t flags, uint32_t target, struct
                                     drm_modeset_acquire_ctx *ctx)
```

do page flip on target vblank period.

Parameters

```
struct drm_crtc *crtc
    DRM CRTC
```

```
struct drm_framebuffer *fb
    DRM framebuffer
```

```
struct drm_pending_vblank_event *event
    optional DRM event to signal upon completion
```

```
uint32_t flags
    flip flags for non-vblank sync'ed updates
```

```
uint32_t target
    specifying the target vblank period when the flip to take effect
```

```
struct drm_modeset_acquire_ctx *ctx
    lock acquisition context
```

Description

Provides a default *drm_crtc_funcs.page_flip_target* implementation. Similar to *drm_atomic_helper_page_flip()* with extra parameter to specify target vblank period to flip.

Return

Returns 0 on success, negative errno numbers on failure.

```
u32 *drm_atomic_helper_bridge_propagate_bus_fmt(struct drm_bridge *bridge, struct
                                                drm_bridge_state *bridge_state,
                                                struct drm_crtc_state *crtc_state,
                                                struct drm_connector_state
                                                *conn_state, u32 output_fmt,
                                                unsigned int *num_input_fmts)
```

Propagate output format to the input end of a bridge

Parameters

```
struct drm_bridge *bridge
    bridge control structure
```

```
struct drm_bridge_state *bridge_state
    new bridge state
```

```
struct drm_crtc_state *crtc_state
    new CRTC state
```

struct drm_connector_state *conn_state

new connector state

u32 output_fmt

tested output bus format

unsigned int *num_input_fmts

will contain the size of the returned array

Description

This helper is a pluggable implementation of the [drm_bridge_funcs.atomic_get_input_bus_fmts](#) operation for bridges that don't modify the bus configuration between their input and their output. It returns an array of input formats with a single element set to **output_fmt**.

Return

a valid format array of size **num_input_fmts**, or NULL if the allocation failed

5.2.4 Atomic State Reset and Initialization

Both the drm core and the atomic helpers assume that there is always the full and correct atomic software state for all connectors, CRTC's and planes available. Which is a bit a problem on driver load and also after system suspend. One way to solve this is to have a hardware state read-out infrastructure which reconstructs the full software state (e.g. the i915 driver).

The simpler solution is to just reset the software state to everything off, which is easiest to do by calling [drm_mode_config_reset\(\)](#). To facilitate this the atomic helpers provide default reset implementations for all hooks.

On the upside the precise state tracking of atomic simplifies system suspend and resume a lot. For drivers using [drm_mode_config_reset\(\)](#) a complete recipe is implemented in [drm_atomic_helper_suspend\(\)](#) and [drm_atomic_helper_resume\(\)](#). For other drivers the building blocks are split out, see the documentation for these functions.

5.2.5 Atomic State Helper Reference

void __drm_atomic_helper_crtc_state_reset(struct [drm_crtc_state](#) *crtc_state, struct [drm_crtc](#) *crtc)

reset the CRTC state

Parameters

struct drm_crtc_state *crtc_state

atomic CRTC state, must not be NULL

struct drm_crtc *crtc

CRTC object, must not be NULL

Description

Initializes the newly allocated **crtc_state** with default values. This is useful for drivers that subclass the CRTC state.

```
void __drm_atomic_helper_crtc_reset(struct drm_crtc *crtc, struct drm_crtc_state
                                   *crtc_state)
```

reset state on CRTC

Parameters

struct *drm_crtc* *crtc
drm CRTC

struct *drm_crtc_state* *crtc_state
CRTC state to assign

Description

Initializes the newly allocated **crtc_state** and assigns it to the *drm_crtc->state* pointer of **crtc**, usually required when initializing the drivers or when called from the *drm_crtc_funcs.reset* hook.

This is useful for drivers that subclass the CRTC state.

```
void drm_atomic_helper_crtc_reset(struct drm_crtc *crtc)
    default drm_crtc_funcs.reset hook for CRTCs
```

Parameters

struct *drm_crtc* *crtc
drm CRTC

Description

Resets the atomic state for **crtc** by freeing the state pointer (which might be NULL, e.g. at driver load time) and allocating a new empty state object.

```
void __drm_atomic_helper_crtc_duplicate_state(struct drm_crtc *crtc, struct
                                             drm_crtc_state *state)
```

copy atomic CRTC state

Parameters

struct *drm_crtc* *crtc
CRTC object

struct *drm_crtc_state* *state
atomic CRTC state

Description

Copies atomic state from a CRTC's current state and resets inferred values. This is useful for drivers that subclass the CRTC state.

```
struct drm_crtc_state *drm_atomic_helper_crtc_duplicate_state(struct drm_crtc *crtc)
    default state duplicate hook
```

Parameters

struct *drm_crtc* *crtc
drm CRTC

Description

Default CRTC state duplicate hook for drivers which don't have their own subclassed CRTC state structure.

```
void __drm_atomic_helper_crtc_destroy_state(struct drm_crtc_state *state)
    release CRTC state
```

Parameters

struct *drm_crtc_state* *state
CRTC state object to release

Description

Releases all resources stored in the CRTC state without actually freeing the memory of the CRTC state. This is useful for drivers that subclass the CRTC state.

```
void drm_atomic_helper_crtc_destroy_state(struct drm_crtc *crtc, struct drm_crtc_state
                                         *state)

    default state destroy hook
```

Parameters

struct *drm_crtc* *crtc
drm CRTC

struct *drm_crtc_state* *state
CRTC state object to release

Description

Default CRTC state destroy hook for drivers which don't have their own subclassed CRTC state structure.

```
void __drm_atomic_helper_plane_state_reset(struct drm_plane_state *plane_state, struct
                                           drm_plane *plane)

    resets plane state to default values
```

Parameters

struct *drm_plane_state* *plane_state
atomic plane state, must not be NULL

struct *drm_plane* *plane
plane object, must not be NULL

Description

Initializes the newly allocated **plane_state** with default values. This is useful for drivers that subclass the CRTC state.

```
void __drm_atomic_helper_plane_reset(struct drm_plane *plane, struct drm_plane_state
                                     *plane_state)

    reset state on plane
```

Parameters

struct *drm_plane* *plane
drm plane

struct *drm_plane_state* *plane_state
plane state to assign

Description

Initializes the newly allocated **plane_state** and assigns it to the *drm_crtc->state* pointer of **plane**, usually required when initializing the drivers or when called from the *drm_plane_funcs.reset* hook.

This is useful for drivers that subclass the plane state.

```
void drm_atomic_helper_plane_reset(struct drm_plane *plane)
    default drm_plane_funcs.reset hook for planes
```

Parameters

```
struct drm_plane *plane
    drm plane
```

Description

Resets the atomic state for **plane** by freeing the state pointer (which might be NULL, e.g. at driver load time) and allocating a new empty state object.

```
void __drm_atomic_helper_plane_duplicate_state(struct drm_plane *plane, struct
                                              drm_plane_state *state)
    copy atomic plane state
```

Parameters

```
struct drm_plane *plane
    plane object
```

```
struct drm_plane_state *state
    atomic plane state
```

Description

Copies atomic state from a plane's current state. This is useful for drivers that subclass the plane state.

```
struct drm_plane_state *drm_atomic_helper_plane_duplicate_state(struct drm_plane
                                                                *plane)
    default state duplicate hook
```

Parameters

```
struct drm_plane *plane
    drm plane
```

Description

Default plane state duplicate hook for drivers which don't have their own subclassed plane state structure.

```
void __drm_atomic_helper_plane_destroy_state(struct drm_plane_state *state)
    release plane state
```

Parameters

```
struct drm_plane_state *state
    plane state object to release
```

Description

Releases all resources stored in the plane state without actually freeing the memory of the plane state. This is useful for drivers that subclass the plane state.

```
void drm_atomic_helper_plane_destroy_state(struct drm_plane *plane, struct  
                                           drm_plane_state *state)
```

default state destroy hook

Parameters

```
struct drm_plane *plane  
    drm plane
```

```
struct drm_plane_state *state  
    plane state object to release
```

Description

Default plane state destroy hook for drivers which don't have their own subclassed plane state structure.

```
void __drm_atomic_helper_connector_state_reset(struct drm_connector_state  
                                              *conn_state, struct drm_connector  
                                              *connector)
```

reset the connector state

Parameters

```
struct drm_connector_state *conn_state  
    atomic connector state, must not be NULL
```

```
struct drm_connector *connector  
    connectotr object, must not be NULL
```

Description

Initializes the newly allocated **conn_state** with default values. This is useful for drivers that subclass the connector state.

```
void __drm_atomic_helper_connector_reset(struct drm_connector *connector, struct  
                                         drm_connector_state *conn_state)
```

reset state on connector

Parameters

```
struct drm_connector *connector  
    drm connector
```

```
struct drm_connector_state *conn_state  
    connector state to assign
```

Description

Initializes the newly allocated **conn_state** and assigns it to the *drm_connector->state* pointer of **connector**, usually required when initializing the drivers or when called from the *drm_connector_funcs.reset* hook.

This is useful for drivers that subclass the connector state.

void **drm_atomic_helper_connector_reset**(struct *drm_connector* *connector)
default *drm_connector_funcs.reset* hook for connectors

Parameters

struct drm_connector *connector
drm connector

Description

Resets the atomic state for **connector** by freeing the state pointer (which might be NULL, e.g. at driver load time) and allocating a new empty state object.

void **drm_atomic_helper_connector_tv_margins_reset**(struct *drm_connector* *connector)
Resets TV connector properties

Parameters

struct drm_connector *connector
DRM connector

Description

Resets the TV-related properties attached to a connector.

void **drm_atomic_helper_connector_tv_reset**(struct *drm_connector* *connector)
Resets Analog TV connector properties

Parameters

struct drm_connector *connector
DRM connector

Description

Resets the analog TV properties attached to a connector

int **drm_atomic_helper_connector_tv_check**(struct *drm_connector* *connector, struct *drm_atomic_state* *state)

Validate an analog TV connector state

Parameters

struct drm_connector *connector
DRM Connector

struct drm_atomic_state *state
the DRM State object

Description

Checks the state object to see if the requested state is valid for an analog TV connector.

Return

0 for success, a negative error code on error.

void **__drm_atomic_helper_connector_duplicate_state**(struct *drm_connector* *connector,
struct *drm_atomic_state* *state)
copy atomic connector state

Parameters

struct `drm_connector` *connector

connector object

struct `drm_connector_state` *state

atomic connector state

Description

Copies atomic state from a connector's current state. This is useful for drivers that subclass the connector state.

```
struct drm_connector_state *drm_atomic_helper_connector_duplicate_state(struct  
                                                                    drm_connector  
                                                                    *connector)
```

default state duplicate hook

Parameters

struct `drm_connector` *connector

drm connector

Description

Default connector state duplicate hook for drivers which don't have their own subclassed connector state structure.

```
void __drm_atomic_helper_connector_destroy_state(struct drm_connector_state *state)  
release connector state
```

Parameters

struct `drm_connector_state` *state

connector state object to release

Description

Releases all resources stored in the connector state without actually freeing the memory of the connector state. This is useful for drivers that subclass the connector state.

```
void drm_atomic_helper_connector_destroy_state(struct drm_connector *connector,  
                                              struct drm_connector_state *state)
```

default state destroy hook

Parameters

struct `drm_connector` *connector

drm connector

struct `drm_connector_state` *state

connector state object to release

Description

Default connector state destroy hook for drivers which don't have their own subclassed connector state structure.

```
void __drm_atomic_helper_private_obj_duplicate_state(struct drm_private_obj *obj,  
                                                    struct drm_private_state *state)
```

copy atomic private state

Parameters

struct drm_private_obj *obj

CRTC object

struct drm_private_state *state

new private object state

Description

Copies atomic state from a private objects's current state and resets inferred values. This is useful for drivers that subclass the private state.

void **__drm_atomic_helper_bridge_duplicate_state**(struct *drm_bridge* *bridge, struct *drm_bridge_state* *state)

Copy atomic bridge state

Parameters

struct drm_bridge *bridge

bridge object

struct drm_bridge_state *state

atomic bridge state

Description

Copies atomic state from a bridge's current state and resets inferred values. This is useful for drivers that subclass the bridge state.

struct *drm_bridge_state* ***drm_atomic_helper_bridge_duplicate_state**(struct *drm_bridge* *bridge)

Duplicate a bridge state object

Parameters

struct drm_bridge *bridge

bridge object

Description

Allocates a new bridge state and initializes it with the current bridge state values. This helper is meant to be used as a bridge *drm_bridge_funcs.atomic_duplicate_state* hook for bridges that don't subclass the bridge state.

void **drm_atomic_helper_bridge_destroy_state**(struct *drm_bridge* *bridge, struct *drm_bridge_state* *state)

Destroy a bridge state object

Parameters

struct drm_bridge *bridge

the bridge this state refers to

struct drm_bridge_state *state

bridge state to destroy

Description

Destroys a bridge state previously created by *drm_atomic_helper_bridge_reset`()* or *:c:type:`drm_atomic_helper_bridge_duplicate_state`()*. This helper is meant to be used as a bridge *:c:type:`drm_bridge_funcs.atomic_destroy_state`* hook for bridges that don't subclass the bridge state.

```
void __drm_atomic_helper_bridge_reset(struct drm_bridge *bridge, struct  
                                     drm_bridge_state *state)
```

Initialize a bridge state to its default

Parameters

struct *drm_bridge* *bridge

the bridge this state refers to

struct *drm_bridge_state* *state

bridge state to initialize

Description

Initializes the bridge state to default values. This is meant to be called by the bridge *drm_bridge_funcs.atomic_reset* hook for bridges that subclass the bridge state.

```
struct drm_bridge_state *drm_atomic_helper_bridge_reset(struct drm_bridge *bridge)
```

Allocate and initialize a bridge state to its default

Parameters

struct *drm_bridge* *bridge

the bridge this state refers to

Description

Allocates the bridge state and initializes it to default values. This helper is meant to be used as a bridge *drm_bridge_funcs.atomic_reset* hook for bridges that don't subclass the bridge state.

5.2.6 GEM Atomic Helper Reference

The GEM atomic helpers library implements generic atomic-commit functions for drivers that use GEM objects. Currently, it provides synchronization helpers, and plane state and framebuffer BO mappings for planes with shadow buffers.

Before scanout, a plane's framebuffer needs to be synchronized with possible writers that draw into the framebuffer. All drivers should call *drm_gem_plane_helper_prepare_fb()* from their implementation of struct *drm_plane_helper.prepare_fb*. It sets the plane's fence from the framebuffer so that the DRM core can synchronize access automatically. *drm_gem_plane_helper_prepare_fb()* can also be used directly as implementation of *prepare_fb*.

```
#include <drm/drm_gem_atomic_helper.h>

struct drm_plane_helper_funcs driver_plane_helper_funcs = {
    ...,
    .prepare_fb = drm_gem_plane_helper_prepare_fb,
};
```

A driver using a shadow buffer copies the content of the shadow buffers into the HW's framebuffer memory during an atomic update. This requires a mapping of the shadow buffer into kernel address space. The mappings cannot be established by commit-tail functions, such as *atomic_update*, as this would violate locking rules around *dma_buf_vmap()*.

The helpers for shadow-buffered planes establish and release mappings, and provide *struct* *drm_shadow_plane_state*, which stores the plane's mapping for commit-tail functions.

Shadow-buffered planes can easily be enabled by using the provided macros `DRM_GEM_SHADOW_PLANE_FUNCS` and `DRM_GEM_SHADOW_PLANE_HELPER_FUNCS`. These macros set up the plane and plane-helper callbacks to point to the shadow-buffer helpers.

```
#include <drm/drm_gem_atomic_helper.h>

struct drm_plane_funcs driver_plane_funcs = {
    ...,
    DRM_GEM_SHADOW_PLANE_FUNCS,
};

struct drm_plane_helper_funcs driver_plane_helper_funcs = {
    ...,
    DRM_GEM_SHADOW_PLANE_HELPER_FUNCS,
};
```

In the driver's atomic-update function, shadow-buffer mappings are available from the plane state. Use *to_drm_shadow_plane_state()* to upcast from *struct* *drm_plane_state*.

```
void driver_plane_atomic_update(struct drm_plane *plane,
                               struct drm_plane_state *old_plane_state)
{
    struct drm_plane_state *plane_state = plane->state;
    struct drm_shadow_plane_state *shadow_plane_state =
        to_drm_shadow_plane_state(plane_state);

    // access shadow buffer via shadow_plane_state->map
}
```

A mapping address for each of the framebuffer's buffer object is stored in *struct* *drm_shadow_plane_state.map*. The mappings are valid while the state is being used.

Drivers that use *struct* *drm_simple_display_pipe* can use `DRM_GEM_SIMPLE_DISPLAY_PIPE_SHADOW_PLANE_FUNCS` to initialize the rsp callbacks. Access to shadow-buffer mappings is similar to regular atomic update.

```
struct drm_simple_display_pipe_funcs driver_pipe_funcs = {
    ...,
    DRM_GEM_SIMPLE_DISPLAY_PIPE_SHADOW_PLANE_FUNCS,
};

void driver_pipe_enable(struct drm_simple_display_pipe *pipe,
                       struct drm_crtc_state *crtc_state,
                       struct drm_plane_state *plane_state)
{
    struct drm_shadow_plane_state *shadow_plane_state =
        to_drm_shadow_plane_state(plane_state);

    // access shadow buffer via shadow_plane_state->map
}
```

DRM_SHADOW_PLANE_MAX_WIDTH

DRM_SHADOW_PLANE_MAX_WIDTH ()

Maximum width of a plane's shadow buffer in pixels

Parameters

Description

For drivers with shadow planes, the maximum width of the framebuffer is usually independent from hardware limitations. Drivers can initialize *struct drm_mode_config*.max_width from DRM_SHADOW_PLANE_MAX_WIDTH.

DRM_SHADOW_PLANE_MAX_HEIGHT

DRM_SHADOW_PLANE_MAX_HEIGHT ()

Maximum height of a plane's shadow buffer in scanlines

Parameters

Description

For drivers with shadow planes, the maximum height of the framebuffer is usually independent from hardware limitations. Drivers can initialize *struct drm_mode_config*.max_height from DRM_SHADOW_PLANE_MAX_HEIGHT.

struct **drm_shadow_plane_state**

plane state for planes with shadow buffers

Definition:

```
struct drm_shadow_plane_state {
    struct drm_plane_state base;
    struct iosys_map map[DRM_FORMAT_MAX_PLANES];
    struct iosys_map data[DRM_FORMAT_MAX_PLANES];
};
```

Members

base

plane state

map

Mappings of the plane's framebuffer BOs in to kernel address space

The memory mappings stored in map should be established in the plane's prepare_fb callback and removed in the cleanup_fb callback.

data

Address of each framebuffer BO's data

The address of the data stored in each mapping. This is different for framebuffers with non-zero offset fields.

Description

For planes that use a shadow buffer, *struct drm_shadow_plane_state* provides the regular plane state plus mappings of the shadow buffer into kernel address space.

struct *drm_shadow_plane_state* *to_drm_shadow_plane_state(struct *drm_plane_state* *state)

upcasts from *struct drm_plane_state*

Parameters

struct drm_plane_state *state

the plane state

DRM_GEM_SHADOW_PLANE_FUNCS

DRM_GEM_SHADOW_PLANE_FUNCS ()

Initializes *struct drm_plane_funcs* for shadow-buffered planes

Parameters

Description

Drivers may use GEM BOs as shadow buffers over the framebuffer memory. This macro initializes *struct drm_plane_funcs* to use the rsp helper functions.

DRM_GEM_SHADOW_PLANE_HELPER_FUNCS

DRM_GEM_SHADOW_PLANE_HELPER_FUNCS ()

Initializes *struct drm_plane_helper_funcs* for shadow-buffered planes

Parameters

Description

Drivers may use GEM BOs as shadow buffers over the framebuffer memory. This macro initializes *struct drm_plane_helper_funcs* to use the rsp helper functions.

DRM_GEM_SIMPLE_DISPLAY_PIPE_SHADOW_PLANE_FUNCS

DRM_GEM_SIMPLE_DISPLAY_PIPE_SHADOW_PLANE_FUNCS ()

Initializes *struct drm_simple_display_pipe_funcs* for shadow-buffered planes

Parameters

Description

Drivers may use GEM BOs as shadow buffers over the framebuffer memory. This macro initializes *struct drm_simple_display_pipe_funcs* to use the rsp helper functions.

int drm_gem_plane_helper_prepare_fb(struct *drm_plane* *plane, struct *drm_plane_state* *state)

Prepare a GEM backed framebuffer

Parameters

struct drm_plane *plane

Plane

struct drm_plane_state *state

Plane state the fence will be attached to

Description

This function extracts the exclusive fence from *drm_gem_object.resv* and attaches it to plane state for the atomic helper to wait on. This is necessary to correctly implement implicit synchronization for any buffers shared as a struct *dma_buf*. This function can be used as the *drm_plane_helper_funcs.prepare_fb* callback.

There is no need for *drm_plane_helper_funcs.cleanup_fb* hook for simple GEM based framebuffer drivers which have their buffers always pinned in memory.

This function is the default implementation for GEM drivers of *drm_plane_helper_funcs.prepare_fb* if no callback is provided.

```
void __drm_gem_duplicate_shadow_plane_state(struct drm_plane *plane, struct  
                                           drm_shadow_plane_state  
                                           *new_shadow_plane_state)
```

 duplicates shadow-buffered plane state

Parameters

struct *drm_plane* *plane
 the plane

struct *drm_shadow_plane_state* *new_shadow_plane_state
 the new shadow-buffered plane state

Description

This function duplicates shadow-buffered plane state. This is helpful for drivers that subclass *struct drm_shadow_plane_state*.

The function does not duplicate existing mappings of the shadow buffers. Mappings are maintained during the atomic commit by the plane's *prepare_fb* and *cleanup_fb* helpers. See *drm_gem_prepare_shadow_fb()* and *drm_gem_cleanup_shadow_fb()* for corresponding helpers.

```
struct drm_plane_state *drm_gem_duplicate_shadow_plane_state(struct drm_plane *plane)  
    duplicates shadow-buffered plane state
```

Parameters

struct *drm_plane* *plane
 the plane

Description

This function implements struct *drm_plane_funcs.atomic_duplicate_state* for shadow-buffered planes. It assumes the existing state to be of type *struct drm_shadow_plane_state* and it allocates the new state to be of this type.

The function does not duplicate existing mappings of the shadow buffers. Mappings are maintained during the atomic commit by the plane's *prepare_fb* and *cleanup_fb* helpers. See *drm_gem_prepare_shadow_fb()* and *drm_gem_cleanup_shadow_fb()* for corresponding helpers.

Return

A pointer to a new plane state on success, or NULL otherwise.

```
void __drm_gem_destroy_shadow_plane_state(struct drm_shadow_plane_state  
                                          *shadow_plane_state)
```

 cleans up shadow-buffered plane state

Parameters

struct drm_shadow_plane_state *shadow_plane_state

the shadow-buffered plane state

Description

This function cleans up shadow-buffered plane state. Helpful for drivers that subclass *struct drm_shadow_plane_state*.

void **drm_gem_destroy_shadow_plane_state**(struct *drm_plane* *plane, struct *drm_plane_state* *plane_state)

deletes shadow-buffered plane state

Parameters

struct drm_plane *plane

the plane

struct drm_plane_state *plane_state

the plane state of type *struct drm_shadow_plane_state*

Description

This function implements struct *drm_plane_funcs.atomic_destroy_state* for shadow-buffered planes. It expects that mappings of shadow buffers have been released already.

void **__drm_gem_reset_shadow_plane**(struct *drm_plane* *plane, struct *drm_shadow_plane_state* *shadow_plane_state)

resets a shadow-buffered plane

Parameters

struct drm_plane *plane

the plane

struct drm_shadow_plane_state *shadow_plane_state

the shadow-buffered plane state

Description

This function resets state for shadow-buffered planes. Helpful for drivers that subclass *struct drm_shadow_plane_state*.

void **drm_gem_reset_shadow_plane**(struct *drm_plane* *plane)

resets a shadow-buffered plane

Parameters

struct drm_plane *plane

the plane

Description

This function implements struct *drm_plane_funcs.reset_plane* for shadow-buffered planes. It assumes the current plane state to be of type struct *drm_shadow_plane* and it allocates the new state of this type.

int **drm_gem_begin_shadow_fb_access**(struct *drm_plane* *plane, struct *drm_plane_state* *plane_state)

prepares shadow framebuffers for CPU access

Parameters

struct drm_plane *plane

the plane

struct drm_plane_state *plane_state

the plane state of type *struct drm_shadow_plane_state*

Description

This function implements struct *drm_plane_helper_funcs.begin_fb_access*. It maps all buffer objects of the plane's framebuffer into kernel address space and stores them in struct *drm_shadow_plane_state.map*. The first data bytes are available in struct *drm_shadow_plane_state.data*.

See *drm_gem_end_shadow_fb_access()* for cleanup.

Return

0 on success, or a negative errno code otherwise.

void **drm_gem_end_shadow_fb_access**(struct *drm_plane* *plane, struct *drm_plane_state* *plane_state)

releases shadow framebuffers from CPU access

Parameters

struct drm_plane *plane

the plane

struct drm_plane_state *plane_state

the plane state of type *struct drm_shadow_plane_state*

Description

This function implements struct *drm_plane_helper_funcs.end_fb_access*. It undoes all effects of *drm_gem_begin_shadow_fb_access()* in reverse order.

See *drm_gem_begin_shadow_fb_access()* for more information.

int **drm_gem_simple_kms_begin_shadow_fb_access**(struct *drm_simple_display_pipe* *pipe, struct *drm_plane_state* *plane_state)

prepares shadow framebuffers for CPU access

Parameters

struct drm_simple_display_pipe *pipe

the simple display pipe

struct drm_plane_state *plane_state

the plane state of type *struct drm_shadow_plane_state*

Description

This function implements struct *drm_simple_display_funcs.begin_fb_access*.

See *drm_gem_begin_shadow_fb_access()* for details and *drm_gem_simple_kms_cleanup_shadow_fb()* for cleanup.

Return

0 on success, or a negative errno code otherwise.

void **drm_gem_simple_kms_end_shadow_fb_access**(struct *drm_simple_display_pipe* *pipe,
struct *drm_plane_state* *plane_state)

releases shadow framebuffers from CPU access

Parameters

struct drm_simple_display_pipe *pipe
the simple display pipe

struct drm_plane_state *plane_state
the plane state of type *struct drm_shadow_plane_state*

Description

This function implements struct *drm_simple_display_funcs*.*end_fb_access*. It undoes all effects of *drm_gem_simple_kms_begin_shadow_fb_access()* in reverse order.

See *drm_gem_simple_kms_begin_shadow_fb_access()*.

void **drm_gem_simple_kms_reset_shadow_plane**(struct *drm_simple_display_pipe* *pipe)
resets a shadow-buffered plane

Parameters

struct drm_simple_display_pipe *pipe
the simple display pipe

Description

This function implements struct *drm_simple_display_funcs*.*reset_plane* for shadow-buffered planes.

struct *drm_plane_state* ***drm_gem_simple_kms_duplicate_shadow_plane_state**(struct
drm_simple_display_pipe *pipe)

duplicates shadow-buffered plane state

Parameters

struct drm_simple_display_pipe *pipe
the simple display pipe

Description

This function implements struct *drm_simple_display_funcs*.*duplicate_plane_state* for shadow-buffered planes. It does not duplicate existing mappings of the shadow buffers. Mappings are maintained during the atomic commit by the plane's *prepare_fb* and *cleanup_fb* helpers.

Return

A pointer to a new plane state on success, or NULL otherwise.

void **drm_gem_simple_kms_destroy_shadow_plane_state**(struct *drm_simple_display_pipe* *pipe, struct *drm_plane_state* *plane_state)

resets shadow-buffered plane state

Parameters

struct drm_simple_display_pipe *pipe
the simple display pipe

struct `drm_plane_state` *plane_state

the plane state of type *struct `drm_shadow_plane_state`*

Description

This function implements `struct drm_simple_display_funcs.destroy_plane_state` for shadow-buffered planes. It expects that mappings of shadow buffers have been released already.

5.3 Simple KMS Helper Reference

This helper library provides helpers for drivers for simple display hardware.

drm_simple_display_pipe_init() initializes a simple display pipeline which has only one full-screen scanout buffer feeding one output. The pipeline is represented by *struct `drm_simple_display_pipe`* and binds together *drm_plane*, *drm_crtc* and *drm_encoder* structures into one fixed entity. Some flexibility for code reuse is provided through a separately allocated *drm_connector* object and supporting optional *drm_bridge* encoder drivers.

Many drivers require only a very simple encoder that fulfills the minimum requirements of the display pipeline and does not add additional functionality. The function *drm_simple_encoder_init()* provides an implementation of such an encoder.

struct `drm_simple_display_pipe_funcs`

helper operations for a simple display pipeline

Definition:

```
struct drm_simple_display_pipe_funcs {
    enum drm_mode_status (*mode_valid)(struct drm_simple_display_pipe *pipe,
    ↪const struct drm_display_mode *mode);
    void (*enable)(struct drm_simple_display_pipe *pipe, struct drm_crtc_state
    ↪*crtc_state, struct drm_plane_state *plane_state);
    void (*disable)(struct drm_simple_display_pipe *pipe);
    int (*check)(struct drm_simple_display_pipe *pipe, struct drm_plane_state
    ↪*plane_state, struct drm_crtc_state *crtc_state);
    void (*update)(struct drm_simple_display_pipe *pipe, struct drm_plane_
    ↪state *old_plane_state);
    int (*prepare_fb)(struct drm_simple_display_pipe *pipe, struct drm_plane_
    ↪state *plane_state);
    void (*cleanup_fb)(struct drm_simple_display_pipe *pipe, struct drm_plane_
    ↪state *plane_state);
    int (*begin_fb_access)(struct drm_simple_display_pipe *pipe, struct drm_
    ↪plane_state *new_plane_state);
    void (*end_fb_access)(struct drm_simple_display_pipe *pipe, struct drm_
    ↪plane_state *plane_state);
    int (*enable_vblank)(struct drm_simple_display_pipe *pipe);
    void (*disable_vblank)(struct drm_simple_display_pipe *pipe);
    void (*reset_crtc)(struct drm_simple_display_pipe *pipe);
    struct drm_crtc_state * (*duplicate_crtc_state)(struct drm_simple_display_
    ↪pipe *pipe);
    void (*destroy_crtc_state)(struct drm_simple_display_pipe *pipe, struct
    ↪drm_crtc_state *crtc_state);
    void (*reset_plane)(struct drm_simple_display_pipe *pipe);
```

```

    struct drm_plane_state * (*duplicate_plane_state)(struct drm_simple_
→display_pipe *pipe);
    void (*destroy_plane_state)(struct drm_simple_display_pipe *pipe, struct_
→drm_plane_state *plane_state);
};

```

Members

mode_valid

This callback is used to check if a specific mode is valid in the crtc used in this simple display pipe. This should be implemented if the display pipe has some sort of restriction in the modes it can display. For example, a given display pipe may be responsible to set a clock value. If the clock can not produce all the values for the available modes then this callback can be used to restrict the number of modes to only the ones that can be displayed. Another reason can be bandwidth mitigation: the memory port on the display controller can have bandwidth limitations not allowing pixel data to be fetched at any rate.

This hook is used by the probe helpers to filter the mode list in [drm_helper_probe_single_connector_modes\(\)](#), and it is used by the atomic helpers to validate modes supplied by userspace in [drm_atomic_helper_check_modeset\(\)](#).

This function is optional.

NOTE:

Since this function is both called from the check phase of an atomic commit, and the mode validation in the probe paths it is not allowed to look at anything else but the passed-in mode, and validate it against configuration-invariant hardware constraints.

RETURNS:

drm_mode_status Enum

enable

This function should be used to enable the pipeline. It is called when the underlying crtc is enabled. This hook is optional.

disable

This function should be used to disable the pipeline. It is called when the underlying crtc is disabled. This hook is optional.

check

This function is called in the check phase of an atomic update, specifically when the underlying plane is checked. The simple display pipeline helpers already check that the plane is not scaled, fills the entire visible area and is always enabled when the crtc is also enabled. This hook is optional.

RETURNS:

0 on success, -EINVAL if the state or the transition can't be supported, -ENOMEM on memory allocation failure and -EDEADLK if an attempt to obtain another state object ran into a [drm_modeset_lock](#) deadlock.

update

This function is called when the underlying plane state is updated. This hook is optional.

This is the function drivers should submit the [drm_pending_vblank_event](#) from. Using either [drm_crtc_arm_vblank_event\(\)](#), when the driver supports vblank interrupt handling,

or `drm_crtc_send_vblank_event()` for more complex case. In case the hardware lacks vblank support entirely, drivers can set `struct drm_crtc_state.no_vblank` in `struct drm_simple_display_pipe_funcs`.check and let DRM's atomic helper fake a vblank event.

prepare_fb

Optional, called by `drm_plane_helper_funcs.prepare_fb`. Please read the documentation for the `drm_plane_helper_funcs.prepare_fb` hook for more details.

For GEM drivers who neither have a **prepare_fb** nor **cleanup_fb** hook set, `drm_gem_plane_helper_prepare_fb()` is called automatically to implement this. Other drivers which need additional plane processing can call `drm_gem_plane_helper_prepare_fb()` from their **prepare_fb** hook.

cleanup_fb

Optional, called by `drm_plane_helper_funcs.cleanup_fb`. Please read the documentation for the `drm_plane_helper_funcs.cleanup_fb` hook for more details.

begin_fb_access

Optional, called by `drm_plane_helper_funcs.begin_fb_access`. Please read the documentation for the `drm_plane_helper_funcs.begin_fb_access` hook for more details.

end_fb_access

Optional, called by `drm_plane_helper_funcs.end_fb_access`. Please read the documentation for the `drm_plane_helper_funcs.end_fb_access` hook for more details.

enable_vblank

Optional, called by `drm_crtc_funcs.enable_vblank`. Please read the documentation for the `drm_crtc_funcs.enable_vblank` hook for more details.

disable_vblank

Optional, called by `drm_crtc_funcs.disable_vblank`. Please read the documentation for the `drm_crtc_funcs.disable_vblank` hook for more details.

reset_crtc

Optional, called by `drm_crtc_funcs.reset`. Please read the documentation for the `drm_crtc_funcs.reset` hook for more details.

duplicate_crtc_state

Optional, called by `drm_crtc_funcs.atomic_duplicate_state`. Please read the documentation for the `drm_crtc_funcs.atomic_duplicate_state` hook for more details.

destroy_crtc_state

Optional, called by `drm_crtc_funcs.atomic_destroy_state`. Please read the documentation for the `drm_crtc_funcs.atomic_destroy_state` hook for more details.

reset_plane

Optional, called by `drm_plane_funcs.reset`. Please read the documentation for the `drm_plane_funcs.reset` hook for more details.

duplicate_plane_state

Optional, called by `drm_plane_funcs.atomic_duplicate_state`. Please read the documentation for the `drm_plane_funcs.atomic_duplicate_state` hook for more details.

destroy_plane_state

Optional, called by `drm_plane_funcs.atomic_destroy_state`. Please read the documentation for the `drm_plane_funcs.atomic_destroy_state` hook for more details.

struct **drm_simple_display_pipe**
simple display pipeline

Definition:

```
struct drm_simple_display_pipe {  
    struct drm_crtc crtc;  
    struct drm_plane plane;  
    struct drm_encoder encoder;  
    struct drm_connector *connector;  
    const struct drm_simple_display_pipe_funcs *funcs;  
};
```

Members

crtc

CRTC control structure

plane

Plane control structure

encoder

Encoder control structure

connector

Connector control structure

funcs

Pipeline control functions (optional)

Description

Simple display pipeline with plane, crtc and encoder collapsed into one entity. It should be initialized by calling *drm_simple_display_pipe_init()*.

drmm_simple_encoder_alloc

drmm_simple_encoder_alloc (dev, type, member, encoder_type)

Allocate and initialize an encoder with basic functionality.

Parameters

dev

drm device

type

the type of the struct which contains struct *drm_encoder*

member

the name of the *drm_encoder* within **type**.

encoder_type

user visible type of the encoder

Description

Allocates and initializes an encoder that has no further functionality. Settings for possible CRTC and clones are left to their initial values. Cleanup is automatically handled through registering *drm_encoder_cleanup()* with *drmm_add_action()*.

Return

Pointer to new encoder, or ERR_PTR on failure.

```
int drm_simple_encoder_init(struct drm_device *dev, struct drm_encoder *encoder, int
                           encoder_type)
```

Initialize a preallocated encoder with basic functionality.

Parameters

```
struct drm_device *dev
    drm device
```

```
struct drm_encoder *encoder
    the encoder to initialize
```

```
int encoder_type
    user visible type of the encoder
```

Description

Initialises a preallocated encoder that has no further functionality. Settings for possible CRTC and clones are left to their initial values. The encoder will be cleaned up automatically as part of the mode-setting cleanup.

The caller of *drm_simple_encoder_init()* is responsible for freeing the encoder's memory after the encoder has been cleaned up. At the moment this only works reliably if the encoder data structure is stored in the device structure. Free the encoder's memory as part of the device release function.

Note

consider using *drm_simple_encoder_alloc()* instead of *drm_simple_encoder_init()* to let the DRM managed resource infrastructure take care of cleanup and deallocation.

Return

Zero on success, error code on failure.

```
int drm_simple_display_pipe_attach_bridge(struct drm_simple_display_pipe *pipe, struct
                                          drm_bridge *bridge)
```

Attach a bridge to the display pipe

Parameters

```
struct drm_simple_display_pipe *pipe
    simple display pipe object
```

```
struct drm_bridge *bridge
    bridge to attach
```

Description

Makes it possible to still use the *drm_simple_display_pipe* helpers when a DRM bridge has to be used.

Note that you probably want to initialize the pipe by passing a NULL connector to *drm_simple_display_pipe_init()*.

Return

Zero on success, negative error code on failure.


```
int drm_simple_display_pipe_init(struct drm_device *dev, struct drm_simple_display_pipe
                                *pipe, const struct drm_simple_display_pipe_funcs
                                *funcs, const uint32_t *formats, unsigned int
                                format_count, const uint64_t *format_modifiers, struct
                                drm_connector *connector)
```

Initialize a simple display pipeline

Parameters

struct *drm_device* *dev
DRM device

struct *drm_simple_display_pipe* *pipe
simple display pipe object to initialize

const struct *drm_simple_display_pipe_funcs* *funcs
callbacks for the display pipe (optional)

const uint32_t *formats
array of supported formats (DRM_FORMAT_*)

unsigned int format_count
number of elements in **formats**

const uint64_t *format_modifiers
array of formats modifiers

struct *drm_connector* *connector
connector to attach and register (optional)

Description

Sets up a display pipeline which consist of a really simple plane-crtc-encoder pipe.

If a connector is supplied, the pipe will be coupled with the provided connector. You may supply a NULL connector when using *drm bridges*, that handle connectors themselves (see *drm_simple_display_pipe_attach_bridge()*).

Teardown of a simple display pipe is all handled automatically by the *drm core* through calling *drm_mode_config_cleanup()*. Drivers afterwards need to release the memory for the structure themselves.

Return

Zero on success, negative error code on failure.

5.4 fbdev Helper Functions Reference

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

Drivers that support a dumb buffer with a virtual address and mmap support, should try out the generic fbdev emulation using *drm_fbdev_generic_setup()*. It will automatically set up deferred I/O if the driver requires a shadow buffer.

Existing fbdev implementations should restore the fbdev console by using *drm_fb_helper_lastclose()* as their *drm_driver.lastclose* callback. They should

also notify the fb helper code from updates to the output configuration by using `drm_fb_helper_output_poll_changed()` as their `drm_mode_config_funcs.output_poll_changed` callback. New implementations of fbdev should be build on top of struct `drm_client_funcs`, which handles this automatically. Setting the old callbacks should be avoided.

For suspend/resume consider using `drm_mode_config_helper_suspend()` and `drm_mode_config_helper_resume()` which takes care of fbdev as well.

All other functions exported by the fb helper library can be used to implement the fbdev driver interface by the driver.

It is possible, though perhaps somewhat tricky, to implement race-free hotplug detection using the fbdev helpers. The `drm_fb_helper_prepare()` helper must be called first to initialize the minimum required to make hotplug detection work. Drivers also need to make sure to properly set up the `drm_mode_config.funcs` member. After calling `drm_kms_helper_poll_init()` it is safe to enable interrupts and start processing hotplug events. At the same time, drivers should initialize all modeset objects such as CRTC's, encoders and connectors. To finish up the fbdev helper initialization, the `drm_fb_helper_init()` function is called. To probe for all attached displays and set up an initial configuration using the detected hardware, drivers should call `drm_fb_helper_initial_config()`.

If `drm_framebuffer_funcs.dirty` is set, the `drm_fb_helper_{cfb,sys}_{write,fillrect,copyarea,imageblt}` functions will accumulate changes and schedule `drm_fb_helper.dirty_work` to run right away. This worker then calls the `dirty()` function ensuring that it will always run in process context since the `fb_*`() function could be running in atomic context. If `drm_fb_helper_deferred_io()` is used as the `deferred_io` callback it will also schedule `dirty_work` with the damage collected from the mmap page writes.

Deferred I/O is not compatible with SHMEM. Such drivers should request an fbdev shadow buffer and call `drm_fbdev_generic_setup()` instead.

struct **drm_fb_helper_surface_size**

describes fbdev size and scanout surface size

Definition:

```
struct drm_fb_helper_surface_size {
    u32 fb_width;
    u32 fb_height;
    u32 surface_width;
    u32 surface_height;
    u32 surface_bpp;
    u32 surface_depth;
};
```

Members

fb_width

fbdev width

fb_height

fbdev height

surface_width

scanout buffer width

surface_height

scanout buffer height

surface_bpp

scanout buffer bpp

surface_depth

scanout buffer depth

Description

Note that the scanout surface width/height may be larger than the fbdev width/height. In case of multiple displays, the scanout surface is sized according to the largest width/height (so it is large enough for all CRTC's to scanout). But the fbdev width/height is sized to the minimum width/ height of all the displays. This ensures that fbcon fits on the smallest of the attached displays. fb_width/fb_height is used by [drm_fb_helper_fill_info\(\)](#) to fill out the fb_info.var structure.

struct drm_fb_helper_funcs

driver callbacks for the fbdev emulation library

Definition:

```
struct drm_fb_helper_funcs {
    int (*fb_probe)(struct drm_fb_helper *helper, struct drm_fb_helper_surface_
→size *sizes);
    int (*fb_dirty)(struct drm_fb_helper *helper, struct drm_clip_rect *clip);
};
```

Members**fb_probe**

Driver callback to allocate and initialize the fbdev info structure. Furthermore it also needs to allocate the DRM framebuffer used to back the fbdev.

This callback is mandatory.

RETURNS:

The driver should return 0 on success and a negative error code on failure.

fb_dirty

Driver callback to update the framebuffer memory. If set, fbdev emulation will invoke this callback in regular intervals after the framebuffer has been written.

This callback is optional.

Returns: 0 on success, or an error code otherwise.

Description

Driver callbacks used by the fbdev emulation helper library.

struct drm_fb_helper

main structure to emulate fbdev on top of KMS

Definition:

```
struct drm_fb_helper {
    struct drm_client_dev client;
    struct drm_client_buffer *buffer;
    struct drm_framebuffer *fb;
    struct drm_device *dev;
    const struct drm_fb_helper_funcs *funcs;
    struct fb_info *info;
    u32 pseudo_palette[17];
    struct drm_clip_rect damage_clip;
    spinlock_t damage_lock;
    struct work_struct damage_work;
    struct work_struct resume_work;
    struct mutex lock;
    struct list_head kernel_fb_list;
    bool delayed_hotplug;
    bool deferred_setup;
    int preferred_bpp;
#ifdef CONFIG_FB_DEFERRED_IO;
    struct fb_deferred_io fbdefio;
#endif;
};
```

Members

client

DRM client used by the generic fbdev emulation.

buffer

Framebuffer used by the generic fbdev emulation.

fb

Scanout framebuffer object

dev

DRM device

funcs

driver callbacks for fb helper

info

emulated fbdev device info struct

pseudo_palette

fake palette of 16 colors

damage_clip

clip rectangle used with `deferred_io` to accumulate damage to the screen buffer

damage_lock

spinlock protecting **damage_clip**

damage_work

worker used to flush the framebuffer

resume_work

worker used during resume if the console lock is already taken

lock

Top-level FBDEV helper lock. This protects all internal data structures and lists, such as **connector_info** and **crtc_info**.

FIXME: fbdev emulation locking is a mess and long term we want to protect all helper internal state with this lock as well as reduce core KMS locking as much as possible.

kernel_fb_list

Entry on the global `kernel_fb_helper_list`, used for kgdb entry/exit.

delayed_hotplug

A hotplug was received while fbdev wasn't in control of the DRM device, i.e. another KMS master was active. The output configuration needs to be reprobe when fbdev is in control again.

deferred_setup

If no outputs are connected (disconnected or unknown) the FB helper code will defer setup until at least one of the outputs shows up. This field keeps track of the status so that setup can be retried at every hotplug event until it succeeds eventually.

Protected by **lock**.

preferred_bpp

Temporary storage for the driver's preferred BPP setting passed to FB helper initialization. This needs to be tracked so that deferred FB helper setup can pass this on.

See also: **deferred_setup**

fbdefio

Temporary storage for the driver's FB deferred I/O handler. If the driver uses the DRM fbdev emulation layer, this is set by the core to a generic deferred I/O handler if a driver is preferring to use a shadow buffer.

Description

This is the main structure used by the fbdev helpers. Drivers supporting fbdev emulation should embedded this into their overall driver structure. Drivers must also fill out a *struct `drm_fb_helper_funcs`* with a few operations.

DRM_FB_HELPER_DEFAULT_OPS

`DRM_FB_HELPER_DEFAULT_OPS ()`

helper define for drm drivers

Parameters**Description**

Helper define to register default implementations of `drm_fb_helper` functions. To be used in `struct fb_ops` of drm drivers.

int **drm_fb_helper_debug_enter**(struct fb_info *info)

implementation for `fb_ops.fb_debug_enter`

Parameters

struct fb_info *info

fbdev registered by the helper

int **drm_fb_helper_debug_leave**(struct fb_info *info)
implementation for fb_ops.fb_debug_leave

Parameters

struct fb_info *info
fbdev registered by the helper

int **drm_fb_helper_restore_fbdev_mode_unlocked**(struct *drm_fb_helper* *fb_helper)
restore fbdev configuration

Parameters

struct drm_fb_helper *fb_helper
driver-allocated fbdev helper, can be NULL

Description

This should be called from driver's drm *drm_driver.lastclose* callback when implementing an fbcon on top of kms using this helper. This ensures that the user isn't greeted with a black screen when e.g. X dies.

Return

Zero if everything went ok, negative error code otherwise.

int **drm_fb_helper_blank**(int blank, struct fb_info *info)
implementation for fb_ops.fb_blank

Parameters

int blank
desired blanking state

struct fb_info *info
fbdev registered by the helper

void **drm_fb_helper_prepare**(struct *drm_device* *dev, struct *drm_fb_helper* *helper, unsigned int preferred_bpp, const struct *drm_fb_helper_funcs* *funcs)
setup a drm_fb_helper structure

Parameters

struct drm_device *dev
DRM device

struct drm_fb_helper *helper
driver-allocated fbdev helper structure to set up

unsigned int preferred_bpp
Preferred bits per pixel for the device.

const struct drm_fb_helper_funcs *funcs
pointer to structure of functions associate with this helper

Description

Sets up the bare minimum to make the framebuffer helper usable. This is useful to implement race-free initialization of the polling helpers.

void **drm_fb_helper_unprepare**(struct *drm_fb_helper* *fb_helper)
clean up a *drm_fb_helper* structure

Parameters

struct *drm_fb_helper* *fb_helper
driver-allocated fbdev helper structure to set up

Description

Cleans up the framebuffer helper. Inverse of *drm_fb_helper_prepare()*.

int **drm_fb_helper_init**(struct *drm_device* *dev, struct *drm_fb_helper* *fb_helper)
initialize a *struct drm_fb_helper*

Parameters

struct *drm_device* *dev
drm device

struct *drm_fb_helper* *fb_helper
driver-allocated fbdev helper structure to initialize

Description

This allocates the structures for the fbdev helper with the given limits. Note that this won't yet touch the hardware (through the driver interfaces) nor register the fbdev. This is only done in *drm_fb_helper_initial_config()* to allow driver writes more control over the exact init sequence.

Drivers must call *drm_fb_helper_prepare()* before calling this function.

Return

Zero if everything went ok, nonzero otherwise.

struct fb_info ***drm_fb_helper_alloc_info**(struct *drm_fb_helper* *fb_helper)
allocate fb_info and some of its members

Parameters

struct *drm_fb_helper* *fb_helper
driver-allocated fbdev helper

Description

A helper to alloc fb_info and the member cmap. Called by the driver within the fb_probe fb_helper callback function. Drivers do not need to release the allocated fb_info structure themselves, this is automatically done when calling *drm_fb_helper_fini()*.

Return

fb_info pointer if things went okay, pointer containing error code otherwise

void **drm_fb_helper_release_info**(struct *drm_fb_helper* *fb_helper)
release fb_info and its members

Parameters

struct *drm_fb_helper* *fb_helper
driver-allocated fbdev helper

Description

A helper to release `fb_info` and the member `cmap`. Drivers do not need to release the allocated `fb_info` structure themselves, this is automatically done when calling `drm_fb_helper_fini()`.

void **drm_fb_helper_unregister_info**(struct *drm_fb_helper* *fb_helper)
unregister fb_info framebuffer device

Parameters

struct drm_fb_helper *fb_helper
driver-allocated fbdev helper, can be NULL

Description

A wrapper around `unregister_framebuffer`, to release the `fb_info` framebuffer device. This must be called before releasing all resources for **fb_helper** by calling `drm_fb_helper_fini()`.

void **drm_fb_helper_fini**(struct *drm_fb_helper* *fb_helper)
finalize a *struct drm_fb_helper*

Parameters

struct drm_fb_helper *fb_helper
driver-allocated fbdev helper, can be NULL

Description

This cleans up all remaining resources associated with **fb_helper**.

void **drm_fb_helper_deferred_io**(struct fb_info *info, struct list_head *pagereflist)
fbdev deferred_io callback function

Parameters

struct fb_info *info
fb_info struct pointer

struct list_head *pagereflist
list of mmap framebuffer pages that have to be flushed

Description

This function is used as the `fb_deferred_io.deferred_io` callback function for flushing the fbdev mmap writes.

void **drm_fb_helper_set_suspend**(struct *drm_fb_helper* *fb_helper, bool suspend)
wrapper around `fb_set_suspend`

Parameters

struct drm_fb_helper *fb_helper
driver-allocated fbdev helper, can be NULL

bool suspend
whether to suspend or resume

Description

A wrapper around `fb_set_suspend` implemented by fbdev core. Use `drm_fb_helper_set_suspend_unlocked()` if you don't need to take the lock yourself

void **drm_fb_helper_set_suspend_unlocked**(struct *drm_fb_helper* *fb_helper, bool suspend)
wrapper around fb_set_suspend that also takes the console lock

Parameters

struct drm_fb_helper *fb_helper
driver-allocated fbdev helper, can be NULL

bool suspend
whether to suspend or resume

Description

A wrapper around fb_set_suspend() that takes the console lock. If the lock isn't available on resume, a worker is tasked with waiting for the lock to become available. The console lock can be pretty contented on resume due to all the printk activity.

This function can be called multiple times with the same state since fb_info.state is checked to see if fbdev is running or not before locking.

Use *drm_fb_helper_set_suspend()* if you need to take the lock yourself.

int **drm_fb_helper_setcmap**(struct fb_cmap *cmap, struct fb_info *info)
implementation for fb_ops.fb_setcmap

Parameters

struct fb_cmap *cmap
cmap to set

struct fb_info *info
fbdev registered by the helper

int **drm_fb_helper_ioctl**(struct fb_info *info, unsigned int cmd, unsigned long arg)
legacy ioctl implementation

Parameters

struct fb_info *info
fbdev registered by the helper

unsigned int cmd
ioctl command

unsigned long arg
ioctl argument

Description

A helper to implement the standard fbdev ioctl. Only FBIO_WAITFORVSYNC is implemented for now.

int **drm_fb_helper_check_var**(struct fb_var_screeninfo *var, struct fb_info *info)
implementation for fb_ops.fb_check_var

Parameters

struct fb_var_screeninfo *var
screeninfo to check

struct fb_info *info
fbdev registered by the helper

int **drm_fb_helper_set_par**(struct fb_info *info)
implementation for fb_ops.fb_set_par

Parameters

struct fb_info *info
fbdev registered by the helper

Description

This will let fbcon do the mode init and is called at initialization time by the fbdev core when registering the driver, and later on through the hotplug callback.

int **drm_fb_helper_pan_display**(struct fb_var_screeninfo *var, struct fb_info *info)
implementation for fb_ops.fb_pan_display

Parameters

struct fb_var_screeninfo *var
updated screen information

struct fb_info *info
fbdev registered by the helper

void **drm_fb_helper_fill_info**(struct fb_info *info, struct *drm_fb_helper* *fb_helper, struct *drm_fb_helper_surface_size* *sizes)
initializes fbdev information

Parameters

struct fb_info *info
fbdev instance to set up

struct drm_fb_helper *fb_helper
fb helper instance to use as template

struct drm_fb_helper_surface_size *sizes
describes fbdev size and scanout surface size

Description

Sets up the variable and fixed fbdev metainformation from the given fb helper instance and the drm framebuffer allocated in *drm_fb_helper.fb*.

Drivers should call this (or their equivalent setup code) from their *drm_fb_helper_funcs.fb_probe* callback after having allocated the fbdev backing storage framebuffer.

int **drm_fb_helper_initial_config**(struct *drm_fb_helper* *fb_helper)
setup a sane initial connector configuration

Parameters

struct drm_fb_helper *fb_helper
fb_helper device struct

Description

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

Note that this also registers the fbdev and so allows userspace to call into the driver through the fbdev interfaces.

This function will call down into the `drm_fb_helper_funcs.fb_probe` callback to let the driver allocate and initialize the fbdev info structure and the drm framebuffer used to back the fbdev. `drm_fb_helper_fill_info()` is provided as a helper to setup simple default values for the fbdev info structure.

HANG DEBUGGING:

When you have fbcon support built-in or already loaded, this function will do a full modeset to setup the fbdev console. Due to locking misdesign in the VT/fbdev subsystem that entire modeset sequence has to be done while holding `console_lock`. Until `console_unlock` is called no dmesg lines will be sent out to consoles, not even serial console. This means when your driver crashes, you will see absolutely nothing else but a system stuck in this function, with no further output. Any kind of `printk()` you place within your own driver or in the drm core modeset code will also never show up.

Standard debug practice is to run the fbcon setup without taking the `console_lock` as a hack, to be able to see backtraces and crashes on the serial line. This can be done by setting the `fb.lockless_register_fb=1` kernel cmdline option.

The other option is to just disable fbdev emulation since very likely the first modeset from userspace will crash in the same way, and is even easier to debug. This can be done by setting the `drm_kms_helper.fbdev_emulation=0` kernel cmdline option.

Return

Zero if everything went ok, nonzero otherwise.

int **drm_fb_helper_hotplug_event**(struct *drm_fb_helper* *fb_helper)
 respond to a hotplug notification by probing all the outputs attached to the fb

Parameters

struct drm_fb_helper *fb_helper
 driver-allocated fbdev helper, can be NULL

Description

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

Called at runtime, takes the mode config locks to be able to check/change the modeset configuration. Must be run from process context (which usually means either the output polling work or a work item launched from the driver's hotplug interrupt).

Note that drivers may call this even before calling `drm_fb_helper_initial_config` but only after `drm_fb_helper_init`. This allows for a race-free fbcon setup and will make sure that the fbdev emulation will not miss any hotplug events.

Return

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

void **drm_fb_helper_lastclose**(struct *drm_device* *dev)
 DRM driver lastclose helper for fbdev emulation

Parameters

struct drm_device *dev
DRM device

Description

This function can be used as the *drm_driver->lastclose* callback for drivers that only need to call *drm_fb_helper_restore_fbdev_mode_unlocked()*.

void **drm_fb_helper_output_poll_changed**(struct *drm_device* *dev)
DRM mode config .output_poll_changed helper for fbdev emulation

Parameters

struct drm_device *dev
DRM device

Description

This function can be used as the *drm_mode_config_funcs.output_poll_changed* callback for drivers that only need to call *drm_fbdev_hotplug_event()*.

void **drm_fbdev_generic_setup**(struct *drm_device* *dev, unsigned int preferred_bpp)
Setup generic fbdev emulation

Parameters

struct drm_device *dev
DRM device

unsigned int preferred_bpp
Preferred bits per pixel for the device.

Description

This function sets up generic fbdev emulation for drivers that supports dumb buffers with a virtual address and that can be mmap'ed. *drm_fbdev_generic_setup()* shall be called after the DRM driver registered the new DRM device with *drm_dev_register()*.

Restore, hotplug events and teardown are all taken care of. Drivers that do suspend/resume need to call *drm_fb_helper_set_suspend_unlocked()* themselves. Simple drivers might use *drm_mode_config_helper_suspend()*.

In order to provide fixed mmap-able memory ranges, generic fbdev emulation uses a shadow buffer in system memory. The implementation blits the shadow fbdev buffer onto the real buffer in regular intervals.

This function is safe to call even when there are no connectors present. Setup will be retried on the next hotplug event.

The fbdev is destroyed by *drm_dev_unregister()*.

5.5 format Helper Functions Reference

unsigned int **drm_fb_clip_offset**(unsigned int pitch, const struct *drm_format_info* *format, const struct *drm_rect* *clip)

Returns the clipping rectangles byte-offset in a framebuffer

Parameters

unsigned int pitch

Framebuffer line pitch in byte

const struct drm_format_info *format

Framebuffer format

const struct drm_rect *clip

Clip rectangle

Return

The byte offset of the clip rectangle's top-left corner within the framebuffer.

void **drm_fb_memcpy**(struct iosys_map *dst, const unsigned int *dst_pitch, const struct iosys_map *src, const struct *drm_framebuffer* *fb, const struct *drm_rect* *clip)

Copy clip buffer

Parameters

struct iosys_map *dst

Array of destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of source buffers

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory. Destination and framebuffer formats must match. No conversion takes place. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

void **drm_fb_swab**(struct iosys_map *dst, const unsigned int *dst_pitch, const struct iosys_map *src, const struct *drm_framebuffer* *fb, const struct *drm_rect* *clip, bool cached)

Swap bytes into clip buffer

Parameters

struct iosys_map *dst

Array of destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of source buffers

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

bool cached

Source buffer is mapped cached (eg. not write-combined)

Description

This function copies parts of a framebuffer to display memory and swaps per-pixel bytes during the process. Destination and framebuffer formats must match. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index. If **cached** is false a temporary buffer is used to cache one pixel line at a time to speed up slow uncached reads.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

```
void drm_fb_xrgb8888_to_rgb332(struct iosys_map *dst, const unsigned int *dst_pitch, const
                               struct iosys_map *src, const struct drm_framebuffer *fb,
                               const struct drm_rect *clip)
```

Convert XRGB8888 to RGB332 clip buffer

Parameters

struct iosys_map *dst

Array of RGB332 destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffers

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. Destination and framebuffer formats must match. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are

planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

Drivers can use this function for RGB332 devices that don't support XRGB8888 natively.

```
void drm_fb_xrgb8888_to_rgb565(struct iosys_map *dst, const unsigned int *dst_pitch, const
                               struct iosys_map *src, const struct drm_framebuffer *fb,
                               const struct drm_rect *clip, bool swab)
```

Convert XRGB8888 to RGB565 clip buffer

Parameters

struct iosys_map *dst

Array of RGB565 destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffer

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

bool swab

Swap bytes

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. Destination and framebuffer formats must match. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

Drivers can use this function for RGB565 devices that don't support XRGB8888 natively.

```
void drm_fb_xrgb8888_to_xrgb1555(struct iosys_map *dst, const unsigned int *dst_pitch,
                                  const struct iosys_map *src, const struct
                                  drm_framebuffer *fb, const struct drm_rect *clip)
```

Convert XRGB8888 to XRGB1555 clip buffer

Parameters

struct iosys_map *dst

Array of XRGB1555 destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffer

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

Drivers can use this function for XRGB1555 devices that don't support XRGB8888 natively.

```
void drm_fb_xrgb8888_to_argb1555(struct iosys_map *dst, const unsigned int *dst_pitch,  
                                const struct iosys_map *src, const struct  
                                drm_framebuffer *fb, const struct drm_rect *clip)
```

Convert XRGB8888 to ARGB1555 clip buffer

Parameters

struct iosys_map *dst

Array of ARGB1555 destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffer

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

Drivers can use this function for ARGB1555 devices that don't support XRGB8888 natively. It sets an opaque alpha channel as part of the conversion.

```
void drm_fb_xrgb8888_to_rgba5551(struct iosys_map *dst, const unsigned int *dst_pitch,  
                                const struct iosys_map *src, const struct  
                                drm_framebuffer *fb, const struct drm_rect *clip)
```

Convert XRGB8888 to RGBA5551 clip buffer

Parameters

struct iosys_map *dst

Array of RGBA5551 destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffer

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

Drivers can use this function for RGBA5551 devices that don't support XRGB8888 natively. It sets an opaque alpha channel as part of the conversion.

```
void drm_fb_xrgb8888_to_rgb888(struct iosys_map *dst, const unsigned int *dst_pitch, const
                               struct iosys_map *src, const struct drm_framebuffer *fb,
                               const struct drm_rect *clip)
```

Convert XRGB8888 to RGB888 clip buffer

Parameters

struct iosys_map *dst

Array of RGB888 destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffers

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. Destination and framebuffer formats must match. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner). Drivers can use this function for RGB888 devices that don't natively support XRGB8888.

```
void drm_fb_xrgb8888_to_argb8888(struct iosys_map *dst, const unsigned int *dst_pitch,  
                                const struct iosys_map *src, const struct  
                                drm_framebuffer *fb, const struct drm_rect *clip)
```

Convert XRGB8888 to ARGB8888 clip buffer

Parameters

struct iosys_map *dst

Array of ARGB8888 destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffer

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

Drivers can use this function for ARGB8888 devices that don't support XRGB8888 natively. It sets an opaque alpha channel as part of the conversion.

```
void drm_fb_xrgb8888_to_xrgb2101010(struct iosys_map *dst, const unsigned int *dst_pitch,  
                                    const struct iosys_map *src, const struct  
                                    drm_framebuffer *fb, const struct drm_rect *clip)
```

Convert XRGB8888 to XRGB2101010 clip buffer

Parameters

struct iosys_map *dst

Array of XRGB2101010 destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffers

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. Destination and framebuffer formats must match. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

Drivers can use this function for XRGB2101010 devices that don't support XRGB8888 natively.

```
void drm_fb_xrgb8888_to_argb2101010(struct iosys_map *dst, const unsigned int *dst_pitch,  
                                   const struct iosys_map *src, const struct  
                                   drm_framebuffer *fb, const struct drm_rect *clip)
```

Convert XRGB8888 to ARGB2101010 clip buffer

Parameters

struct iosys_map *dst

Array of ARGB2101010 destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffers

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

Drivers can use this function for ARGB2101010 devices that don't support XRGB8888 natively.

```
void drm_fb_xrgb8888_to_gray8(struct iosys_map *dst, const unsigned int *dst_pitch, const  
                             struct iosys_map *src, const struct drm_framebuffer *fb,  
                             const struct drm_rect *clip)
```

Convert XRGB8888 to grayscale

Parameters

struct iosys_map *dst

Array of 8-bit grayscale destination buffers

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffers

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. Destination and framebuffer formats must match. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

DRM doesn't have native monochrome or grayscale support. Drivers can use this function for grayscale devices that don't support XRGB8888 natively. Such drivers can announce the commonly supported XR24 format to userspace and use this function to convert to the native format. Monochrome drivers will use the most significant bit, where 1 means foreground color and 0 background color. ITU BT.601 is being used for the RGB -> luma (brightness) conversion.

int **drm_fb_blit**(struct iosys_map *dst, const unsigned int *dst_pitch, uint32_t dst_format, const struct iosys_map *src, const struct *drm_framebuffer* *fb, const struct *drm_rect* *clip)

Copy parts of a framebuffer to display memory

Parameters

struct iosys_map *dst

Array of display-memory addresses to copy to

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

uint32_t dst_format

FOURCC code of the display's color format

const struct iosys_map *src

The framebuffer memory to copy from

const struct drm_framebuffer *fb

The framebuffer to copy from

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory. If the formats of the display and the framebuffer mismatch, the blit function will attempt to convert between them during the

process. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **dst_format**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner).

Return

0 on success, or -EINVAL if the color-format conversion failed, or a negative error code otherwise.

```
void drm_fb_xrgb8888_to_mono(struct iosys_map *dst, const unsigned int *dst_pitch, const
                             struct iosys_map *src, const struct drm_framebuffer *fb,
                             const struct drm_rect *clip)
```

Convert XRGB8888 to monochrome

Parameters

struct iosys_map *dst

Array of monochrome destination buffers (0=black, 1=white)

const unsigned int *dst_pitch

Array of numbers of bytes between the start of two consecutive scanlines within **dst**; can be NULL if scanlines are stored next to each other.

const struct iosys_map *src

Array of XRGB8888 source buffers

const struct drm_framebuffer *fb

DRM framebuffer

const struct drm_rect *clip

Clip rectangle area to copy

Description

This function copies parts of a framebuffer to display memory and converts the color format during the process. Destination and framebuffer formats must match. The parameters **dst**, **dst_pitch** and **src** refer to arrays. Each array must have at least as many entries as there are planes in **fb**'s format. Each entry stores the value for the format's respective color plane at the same index.

This function does not apply clipping on **dst** (i.e. the destination is at the top-left corner). The first pixel (upper left corner of the clip rectangle) will be converted and copied to the first bit (LSB) in the first byte of the monochrome destination buffer. If the caller requires that the first pixel in a byte must be located at an x-coordinate that is a multiple of 8, then the caller must take care itself of supplying a suitable clip rectangle.

DRM doesn't have native monochrome support. Drivers can use this function for monochrome devices that don't support XRGB8888 natively. Such drivers can announce the commonly supported XR24 format to userspace and use this function to convert to the native format.

This function uses *drm_fb_xrgb8888_to_gray8()* to convert to grayscale and then the result is converted from grayscale to monochrome.

```
size_t drm_fb_build_fourcc_list(struct drm_device *dev, const u32 *native_fourccs, size_t
                                native_nfourccs, u32 *fourccs_out, size_t nfourccs_out)
```

Filters a list of supported color formats against the device's native formats

Parameters

struct drm_device *dev
DRM device

const u32 *native_fourccs
4CC codes of natively supported color formats

size_t native_nfourccs
The number of entries in **native_fourccs**

u32 *fourccs_out
Returns 4CC codes of supported color formats

size_t nfourccs_out
The number of available entries in **fourccs_out**

Description

This function create a list of supported color format from natively supported formats and additional emulated formats. At a minimum, most userspace programs expect at least support for XRGB8888 on the primary plane. Devices that have to emulate the format, and possibly others, can use [drm_fb_build_fourcc_list\(\)](#) to create a list of supported color formats. The returned list can be handed over to [drm_universal_plane_init\(\)](#) et al. Native formats will go before emulated formats. Native formats with alpha channel will be replaced by such without, as primary planes usually don't support alpha. Other heuristics might be applied to optimize the order. Formats near the beginning of the list are usually preferred over formats near the end of the list.

Return

The number of color-formats 4CC codes returned in **fourccs_out**.

5.6 Framebuffer DMA Helper Functions Reference

Provides helper functions for creating a DMA-contiguous framebuffer.

Depending on the platform, the buffers may be physically non-contiguous and mapped through an IOMMU or a similar mechanism, or allocated from physically-contiguous memory (using, for instance, CMA or a pool of memory reserved at early boot). This is handled behind the scenes by the DMA mapping API.

[drm_gem_fb_create\(\)](#) is used in the [drm_mode_config_funcs.fb_create](#) callback function to create a DMA-contiguous framebuffer.

struct [drm_gem_dma_object](#) *drm_fb_dma_get_gem_obj (struct [drm_framebuffer](#) *fb,
unsigned int plane)

Get DMA GEM object for framebuffer

Parameters

struct drm_framebuffer *fb
The framebuffer

unsigned int plane
Which plane

Description

Return the DMA GEM object for given framebuffer.

This function will usually be called from the CRTC callback functions.

`dma_addr_t drm_fb_dma_get_gem_addr(struct drm_framebuffer *fb, struct drm_plane_state *state, unsigned int plane)`

Get DMA (bus) address for framebuffer, for pixel formats where values are grouped in blocks this will get you the beginning of the block

Parameters

struct *drm_framebuffer* *fb

The framebuffer

struct *drm_plane_state* *state

Which state of drm plane

unsigned int plane

Which plane Return the DMA GEM address for given framebuffer.

Description

This function will usually be called from the PLANE callback functions.

`void drm_fb_dma_sync_non_coherent(struct drm_device *drm, struct drm_plane_state *old_state, struct drm_plane_state *state)`

Sync GEM object to non-coherent backing memory

Parameters

struct *drm_device* *drm

DRM device

struct *drm_plane_state* *old_state

Old plane state

struct *drm_plane_state* *state

New plane state

Description

This function can be used by drivers that use damage clips and have DMA GEM objects backed by non-coherent memory. Calling this function in a plane's `.atomic_update` ensures that all the data in the backing memory have been written to RAM.

5.7 Framebuffer GEM Helper Reference

This library provides helpers for drivers that don't subclass *drm_framebuffer* and use *drm_gem_object* for their backing storage.

Drivers without additional needs to validate framebuffers can simply use *drm_gem_fb_create()* and everything is wired up automatically. Other drivers can use all parts independently.

`struct drm_gem_object *drm_gem_fb_get_obj(struct drm_framebuffer *fb, unsigned int plane)`

Get GEM object backing the framebuffer

Parameters

struct drm_framebuffer *fb
Framebuffer

unsigned int plane
Plane index

Description

No additional reference is taken beyond the one that the `drm_framebuffer` already holds.

Return

Pointer to [`drm_gem_object`](#) for the given framebuffer and plane index or NULL if it does not exist.

void **drm_gem_fb_destroy**(struct [`drm_framebuffer`](#) *fb)
Free GEM backed framebuffer

Parameters

struct drm_framebuffer *fb
Framebuffer

Description

Frees a GEM backed framebuffer with its backing buffer(s) and the structure itself. Drivers can use this as their [`drm_framebuffer_funcs->destroy`](#) callback.

int **drm_gem_fb_create_handle**(struct [`drm_framebuffer`](#) *fb, struct [`drm_file`](#) *file, unsigned int *handle)
Create handle for GEM backed framebuffer

Parameters

struct drm_framebuffer *fb
Framebuffer

struct drm_file *file
DRM file to register the handle for

unsigned int *handle
Pointer to return the created handle

Description

This function creates a handle for the GEM object backing the framebuffer. Drivers can use this as their [`drm_framebuffer_funcs->create_handle`](#) callback. The GETFB IOCTL calls into this callback.

Return

0 on success or a negative error code on failure.

int **drm_gem_fb_init_with_funcs**(struct [`drm_device`](#) *dev, struct [`drm_framebuffer`](#) *fb, struct [`drm_file`](#) *file, const struct [`drm_mode_fb_cmd2`](#) *mode_cmd, const struct [`drm_framebuffer_funcs`](#) *funcs)
Helper function for implementing [`drm_mode_config_funcs.fb_create`](#) callback in cases when the driver allocates a subclass of [`struct drm_framebuffer`](#)

Parameters

struct *drm_device* *dev
DRM device

struct *drm_framebuffer* *fb
framebuffer object

struct *drm_file* *file
DRM file that holds the GEM handle(s) backing the framebuffer

const struct *drm_mode_fb_cmd2* *mode_cmd
Metadata from the userspace framebuffer creation request

const struct *drm_framebuffer_funcs* *funcs
vtable to be used for the new framebuffer object

Description

This function can be used to set *drm_framebuffer_funcs* for drivers that need custom framebuffer callbacks. Use *drm_gem_fb_create()* if you don't need to change *drm_framebuffer_funcs*. The function does buffer size validation. The buffer size validation is for a general case, though, so users should pay attention to the checks being appropriate for them or, at least, non-conflicting.

Return

Zero or a negative error code.

struct *drm_framebuffer* ***drm_gem_fb_create_with_funcs**(struct *drm_device* *dev, struct *drm_file* *file, const struct *drm_mode_fb_cmd2* *mode_cmd, const struct *drm_framebuffer_funcs* *funcs)

Helper function for the *drm_mode_config_funcs.fb_create* callback

Parameters

struct *drm_device* *dev
DRM device

struct *drm_file* *file
DRM file that holds the GEM handle(s) backing the framebuffer

const struct *drm_mode_fb_cmd2* *mode_cmd
Metadata from the userspace framebuffer creation request

const struct *drm_framebuffer_funcs* *funcs
vtable to be used for the new framebuffer object

Description

This function can be used to set *drm_framebuffer_funcs* for drivers that need custom framebuffer callbacks. Use *drm_gem_fb_create()* if you don't need to change *drm_framebuffer_funcs*. The function does buffer size validation.

Return

Pointer to a *drm_framebuffer* on success or an error pointer on failure.

```
struct drm_framebuffer *drm_gem_fb_create(struct drm_device *dev, struct drm_file *file,  
                                         const struct drm_mode_fb_cmd2 *mode_cmd)
```

Helper function for the *drm_mode_config_funcs.fb_create* callback

Parameters

struct *drm_device* *dev

DRM device

struct *drm_file* *file

DRM file that holds the GEM handle(s) backing the framebuffer

const struct *drm_mode_fb_cmd2* *mode_cmd

Metadata from the userspace framebuffer creation request

Description

This function creates a new framebuffer object described by *drm_mode_fb_cmd2*. This description includes handles for the buffer(s) backing the framebuffer.

If your hardware has special alignment or pitch requirements these should be checked before calling this function. The function does buffer size validation. Use *drm_gem_fb_create_with_dirty()* if you need framebuffer flushing.

Drivers can use this as their *drm_mode_config_funcs.fb_create* callback. The ADDFB2 IOCTL calls into this callback.

Return

Pointer to a *drm_framebuffer* on success or an error pointer on failure.

```
struct drm_framebuffer *drm_gem_fb_create_with_dirty(struct drm_device *dev, struct  
                                                    drm_file *file, const struct  
                                                    drm_mode_fb_cmd2 *mode_cmd)
```

Helper function for the *drm_mode_config_funcs.fb_create* callback

Parameters

struct *drm_device* *dev

DRM device

struct *drm_file* *file

DRM file that holds the GEM handle(s) backing the framebuffer

const struct *drm_mode_fb_cmd2* *mode_cmd

Metadata from the userspace framebuffer creation request

Description

This function creates a new framebuffer object described by *drm_mode_fb_cmd2*. This description includes handles for the buffer(s) backing the framebuffer. *drm_atomic_helper_dirtyfb()* is used for the dirty callback giving framebuffer flushing through the atomic machinery. Use *drm_gem_fb_create()* if you don't need the dirty callback. The function does buffer size validation.

Drivers should also call *drm_plane_enable_fb_damage_clips()* on all planes to enable userspace to use damage clips also with the ATOMIC IOCTL.

Drivers can use this as their *drm_mode_config_funcs.fb_create* callback. The ADDFB2 IOCTL calls into this callback.

Return

Pointer to a [drm_framebuffer](#) on success or an error pointer on failure.

```
int drm_gem_fb_vmap(struct drm\_framebuffer *fb, struct iosys_map *map, struct iosys_map
                    *data)
```

maps all framebuffer BOs into kernel address space

Parameters

struct drm_framebuffer *fb
the framebuffer

struct iosys_map *map
returns the mapping's address for each BO

struct iosys_map *data
returns the data address for each BO, can be NULL

Description

This function maps all buffer objects of the given framebuffer into kernel address space and stores them in struct iosys_map. If the mapping operation fails for one of the BOs, the function unmaps the already established mappings automatically.

Callers that want to access a BO's stored data should pass **data**. The argument returns the addresses of the data stored in each BO. This is different from **map** if the framebuffer's offsets field is non-zero.

Both, **map** and **data**, must each refer to arrays with at least fb->format->num_planes elements.

See [drm_gem_fb_vunmap\(\)](#) for unmapping.

Return

0 on success, or a negative errno code otherwise.

```
void drm_gem_fb_vunmap(struct drm\_framebuffer *fb, struct iosys_map *map)
```

unmaps framebuffer BOs from kernel address space

Parameters

struct drm_framebuffer *fb
the framebuffer

struct iosys_map *map
mapping addresses as returned by [drm_gem_fb_vmap\(\)](#)

Description

This function unmaps all buffer objects of the given framebuffer.

See [drm_gem_fb_vmap\(\)](#) for more information.

```
int drm_gem_fb_begin_cpu_access(struct drm\_framebuffer *fb, enum dma_data_direction
                               dir)
```

prepares GEM buffer objects for CPU access

Parameters

struct drm_framebuffer *fb
the framebuffer

enum dma_data_direction dir
access mode

Description

Prepares a framebuffer's GEM buffer objects for CPU access. This function must be called before accessing the BO data within the kernel. For imported BOs, the function calls `dma_buf_begin_cpu_access()`.

See `drm_gem_fb_end_cpu_access()` for signalling the end of CPU access.

Return

0 on success, or a negative errno code otherwise.

void **drm_gem_fb_end_cpu_access**(struct *drm_framebuffer* *fb, enum dma_data_direction dir)
signals end of CPU access to GEM buffer objects

Parameters

struct drm_framebuffer *fb
the framebuffer

enum dma_data_direction dir
access mode

Description

Signals the end of CPU access to the given framebuffer's GEM buffer objects. This function must be paired with a corresponding call to `drm_gem_fb_begin_cpu_access()`. For imported BOs, the function calls `dma_buf_end_cpu_access()`.

See also `drm_gem_fb_begin_cpu_access()`.

int **drm_gem_fb_afbc_init**(struct *drm_device* *dev, const struct *drm_mode_fb_cmd2* *mode_cmd, struct *drm_afbc_framebuffer* *afbc_fb)

Helper function for drivers using afbc to fill and validate all the afbc-specific *struct drm_afbc_framebuffer* members

Parameters

struct drm_device *dev
DRM device

const struct drm_mode_fb_cmd2 *mode_cmd
Metadata from the userspace framebuffer creation request

struct drm_afbc_framebuffer *afbc_fb
afbc framebuffer

Description

This function can be used by drivers which support afbc to complete the preparation of *struct drm_afbc_framebuffer*. It must be called after allocating the said struct and calling `drm_gem_fb_init_with_funcs()`. It is caller's responsibility to put `afbc_fb->base.obj` objects in case the call is unsuccessful.

Return

Zero on success or a negative error value on failure.

5.8 Bridges

5.8.1 Overview

`struct drm_bridge` represents a device that hangs on to an encoder. These are handy when a regular `drm_encoder` entity isn't enough to represent the entire encoder chain.

A bridge is always attached to a single `drm_encoder` at a time, but can be either connected to it directly, or through a chain of bridges:

```
[ CRTC ---> ] Encoder ---> Bridge A ---> Bridge B
```

Here, the output of the encoder feeds to bridge A, and that further feeds to bridge B. Bridge chains can be arbitrarily long, and shall be fully linear: Chaining multiple bridges to the output of a bridge, or the same bridge to the output of different bridges, is not supported.

`drm_bridge`, like `drm_panel`, aren't `drm_mode_object` entities like planes, CRTC's, encoders or connectors and hence are not visible to userspace. They just provide additional hooks to get the desired output at the end of the encoder chain.

5.8.2 Display Driver Integration

Display drivers are responsible for linking encoders with the first bridge in the chains. This is done by acquiring the appropriate bridge with `devm_drm_of_get_bridge()`. Once acquired, the bridge shall be attached to the encoder with a call to `drm_bridge_attach()`.

Bridges are responsible for linking themselves with the next bridge in the chain, if any. This is done the same way as for encoders, with the call to `drm_bridge_attach()` occurring in the `drm_bridge_funcs.attach` operation.

Once these links are created, the bridges can participate along with encoder functions to perform mode validation and fixup (through `drm_bridge_chain_mode_valid()` and `drm_atomic_bridge_chain_check()`), mode setting (through `drm_bridge_chain_mode_set()`), enable (through `drm_atomic_bridge_chain_pre_enable()` and `drm_atomic_bridge_chain_enable()`) and disable (through `drm_atomic_bridge_chain_disable()` and `drm_atomic_bridge_chain_post_disable()`). Those functions call the corresponding operations provided in `drm_bridge_funcs` in sequence for all bridges in the chain.

For display drivers that use the atomic helpers `drm_atomic_helper_check_modeset()`, `drm_atomic_helper_commit_modeset_enables()` and `drm_atomic_helper_commit_modeset_disables()` (either directly in hand-rolled commit check and commit tail handlers, or through the higher-level `drm_atomic_helper_check()` and `drm_atomic_helper_commit_tail()` or `drm_atomic_helper_commit_tail_rpm()` helpers), this is done transparently and requires no intervention from the driver. For other drivers, the relevant DRM bridge chain functions shall be called manually.

Bridges also participate in implementing the `drm_connector` at the end of the bridge chain. Display drivers may use the `drm_bridge_connector_init()` helper to create the `drm_connector`, or implement it manually on top of the connector-related operations exposed by the bridge (see the overview documentation of bridge operations for more details).

5.8.3 Special Care with MIPI-DSI bridges

The interaction between the bridges and other frameworks involved in the probing of the upstream driver and the bridge driver can be challenging. Indeed, there's multiple cases that needs to be considered:

- The upstream driver doesn't use the component framework and isn't a MIPI-DSI host. In this case, the bridge driver will probe at some point and the upstream driver should try to probe again by returning `EPROBE_DEFER` as long as the bridge driver hasn't probed.
- The upstream driver doesn't use the component framework, but is a MIPI-DSI host. The bridge device uses the MIPI-DCS commands to be controlled. In this case, the bridge device is a child of the display device and when it will probe it's assured that the display device (and MIPI-DSI host) is present. The upstream driver will be assured that the bridge driver is connected between the `mipi_dsi_host_ops.attach` and `mipi_dsi_host_ops.detach` operations. Therefore, it must run `mipi_dsi_host_register()` in its probe function, and then run `drm_bridge_attach()` in its `mipi_dsi_host_ops.attach` hook.
- The upstream driver uses the component framework and is a MIPI-DSI host. The bridge device uses the MIPI-DCS commands to be controlled. This is the same situation than above, and can run `mipi_dsi_host_register()` in either its probe or bind hooks.
- The upstream driver uses the component framework and is a MIPI-DSI host. The bridge device uses a separate bus (such as I2C) to be controlled. In this case, there's no correlation between the probe of the bridge and upstream drivers, so care must be taken to avoid an endless `EPROBE_DEFER` loop, with each driver waiting for the other to probe.

The ideal pattern to cover the last item (and all the others in the MIPI-DSI host driver case) is to split the operations like this:

- The MIPI-DSI host driver must run `mipi_dsi_host_register()` in its probe hook. It will make sure that the MIPI-DSI host sticks around, and that the driver's bind can be called.
- In its probe hook, the bridge driver must try to find its MIPI-DSI host, register as a MIPI-DSI device and attach the MIPI-DSI device to its host. The bridge driver is now functional.
- In its `struct mipi_dsi_host_ops.attach` hook, the MIPI-DSI host can now add its component. Its bind hook will now be called and since the bridge driver is attached and registered, we can now look for and attach it.

At this point, we're now certain that both the upstream driver and the bridge driver are functional and we can't have a deadlock-like situation when probing.

5.8.4 Bridge Operations

Bridge drivers expose operations through the `drm_bridge_funcs` structure. The DRM internals (atomic and CRTC helpers) use the helpers defined in `drm_bridge.c` to call bridge operations. Those operations are divided in three big categories to support different parts of the bridge usage.

- The encoder-related operations support control of the bridges in the chain, and are roughly counterparts to the `drm_encoder_helper_funcs` operations. They are used by the legacy CRTC and the atomic modeset helpers to perform mode validation, fixup and setting, and enable and disable the bridge automatically.

The enable and disable operations are split in `drm_bridge_funcs.pre_enable`, `drm_bridge_funcs.enable`, `drm_bridge_funcs.disable` and `drm_bridge_funcs.post_disable` to provide finer-grained control.

Bridge drivers may implement the legacy version of those operations, or the atomic version (prefixed with `atomic_`), in which case they shall also implement the atomic state book-keeping operations (`drm_bridge_funcs.atomic_duplicate_state`, `drm_bridge_funcs.atomic_destroy_state` and `drm_bridge_funcs.reset`). Mixing atomic and non-atomic versions of the operations is not supported.

- The bus format negotiation operations `drm_bridge_funcs.atomic_get_output_bus_fmts` and `drm_bridge_funcs.atomic_get_input_bus_fmts` allow bridge drivers to negotiate the formats transmitted between bridges in the chain when multiple formats are supported. Negotiation for formats is performed transparently for display drivers by the atomic modeset helpers. Only atomic versions of those operations exist, bridge drivers that need to implement them shall thus also implement the atomic version of the encoder-related operations. This feature is not supported by the legacy CRTC helpers.
- The connector-related operations support implementing a `drm_connector` based on a chain of bridges. DRM bridges traditionally create a `drm_connector` for bridges meant to be used at the end of the chain. This puts additional burden on bridge drivers, especially for bridges that may be used in the middle of a chain or at the end of it. Furthermore, it requires all operations of the `drm_connector` to be handled by a single bridge, which doesn't always match the hardware architecture.

To simplify bridge drivers and make the connector implementation more flexible, a new model allows bridges to unconditionally skip creation of `drm_connector` and instead expose `drm_bridge_funcs` operations to support an externally-implemented `drm_connector`. Those operations are `drm_bridge_funcs.detect`, `drm_bridge_funcs.get_modes`, `drm_bridge_funcs.get_edid`, `drm_bridge_funcs.hpd_notify`, `drm_bridge_funcs.hpd_enable` and `drm_bridge_funcs.hpd_disable`. When implemented, display drivers shall create a `drm_connector` instance for each chain of bridges, and implement those connector instances based on the bridge connector operations.

Bridge drivers shall implement the connector-related operations for all the features that the bridge hardware support. For instance, if a bridge supports reading EDID, the `drm_bridge_funcs.get_edid` shall be implemented. This however doesn't mean that the DDC lines are wired to the bridge on a particular platform, as they could also be connected to an I2C controller of the SoC. Support for the connector-related operations on the running platform is reported through the `drm_bridge.ops` flags. Bridge drivers shall detect which operations they can support on the platform (usually this information is provided by ACPI or DT), and set the `drm_bridge.ops` flags for all supported operations. A flag shall only be set if the corresponding `drm_bridge_funcs` operation is implemented, but an implemented operation doesn't necessarily imply that the corresponding flag will be set. Display drivers shall use the `drm_bridge.ops` flags to decide which bridge to delegate a connector operation to. This mechanism allows providing a single static const `drm_bridge_funcs` instance in bridge drivers, improving security by storing function pointers in read-only memory.

In order to ease transition, bridge drivers may support both the old and new models by making connector creation optional and implementing the connected-related bridge operations. Connector creation is then controlled by the flags argument to the `drm_bridge_attach()` function. Display drivers that support the new model and create connectors themselves shall set the `DRM_BRIDGE_ATTACH_NO_CONNECTOR` flag, and

bridge drivers shall then skip connector creation. For intermediate bridges in the chain, the flag shall be passed to the `drm_bridge_attach()` call for the downstream bridge. Bridge drivers that implement the new model only shall return an error from their `drm_bridge_funcs.attach` handler when the `DRM_BRIDGE_ATTACH_NO_CONNECTOR` flag is not set. New display drivers should use the new model, and convert the bridge drivers they use if needed, in order to gradually transition to the new model.

5.8.5 Bridge Connector Helper

The DRM bridge connector helper object provides a DRM connector implementation that wraps a chain of `struct drm_bridge`. The connector operations are fully implemented based on the operations of the bridges in the chain, and don't require any intervention from the display controller driver at runtime.

To use the helper, display controller drivers create a bridge connector with a call to `drm_bridge_connector_init()`. This associates the newly created connector with the chain of bridges passed to the function and registers it with the DRM device. At that point the connector becomes fully usable, no further operation is needed.

The DRM bridge connector operations are implemented based on the operations provided by the bridges in the chain. Each connector operation is delegated to the bridge closest to the connector (at the end of the chain) that provides the relevant functionality.

To make use of this helper, all bridges in the chain shall report bridge operation flags (`drm_bridge->ops`) and bridge output type (`drm_bridge->type`), as well as the `DRM_BRIDGE_ATTACH_NO_CONNECTOR` attach flag (none of the bridges shall create a DRM connector directly).

5.8.6 Bridge Helper Reference

enum **drm_bridge_attach_flags**

Flags for `drm_bridge_funcs.attach`

Constants

DRM_BRIDGE_ATTACH_NO_CONNECTOR

When this flag is set the bridge shall not create a `drm_connector`.

struct **drm_bridge_funcs**

`drm_bridge` control functions

Definition:

```
struct drm_bridge_funcs {
    int (*attach)(struct drm_bridge *bridge, enum drm_bridge_attach_flags
→ flags);
    void (*detach)(struct drm_bridge *bridge);
    enum drm_mode_status (*mode_valid)(struct drm_bridge *bridge, const struct
→ drm_display_info *info, const struct drm_display_mode *mode);
    bool (*mode_fixup)(struct drm_bridge *bridge, const struct drm_display_mode
→ *mode, struct drm_display_mode *adjusted_mode);
    void (*disable)(struct drm_bridge *bridge);
    void (*post_disable)(struct drm_bridge *bridge);
}
```



```

void (*mode_set)(struct drm_bridge *bridge, const struct drm_display_mode
↪ *mode, const struct drm_display_mode *adjusted_mode);
void (*pre_enable)(struct drm_bridge *bridge);
void (*enable)(struct drm_bridge *bridge);
void (*atomic_pre_enable)(struct drm_bridge *bridge, struct drm_bridge_
↪ state *old_bridge_state);
void (*atomic_enable)(struct drm_bridge *bridge, struct drm_bridge_state
↪ *old_bridge_state);
void (*atomic_disable)(struct drm_bridge *bridge, struct drm_bridge_state
↪ *old_bridge_state);
void (*atomic_post_disable)(struct drm_bridge *bridge, struct drm_bridge_
↪ state *old_bridge_state);
struct drm_bridge_state *(*atomic_duplicate_state)(struct drm_bridge
↪ *bridge);
void (*atomic_destroy_state)(struct drm_bridge *bridge, struct drm_bridge_
↪ state *state);
u32 *(*atomic_get_output_bus_fmts)(struct drm_bridge *bridge, struct drm_
↪ bridge_state *bridge_state, struct drm_crtc_state *crtc_state, struct drm_
↪ connector_state *conn_state, unsigned int *num_output_fmts);
u32 *(*atomic_get_input_bus_fmts)(struct drm_bridge *bridge, struct drm_
↪ bridge_state *bridge_state, struct drm_crtc_state *crtc_state, struct drm_
↪ connector_state *conn_state, u32 output_fmt, unsigned int *num_input_fmts);
int (*atomic_check)(struct drm_bridge *bridge, struct drm_bridge_state
↪ *bridge_state, struct drm_crtc_state *crtc_state, struct drm_connector_state
↪ *conn_state);
struct drm_bridge_state *(*atomic_reset)(struct drm_bridge *bridge);
enum drm_connector_status (*detect)(struct drm_bridge *bridge);
int (*get_modes)(struct drm_bridge *bridge, struct drm_connector
↪ *connector);
const struct drm_edid *(*edid_read)(struct drm_bridge *bridge, struct drm_
↪ connector *connector);
struct edid *(*get_edid)(struct drm_bridge *bridge, struct drm_connector
↪ *connector);
void (*hpd_notify)(struct drm_bridge *bridge, enum drm_connector_status
↪ status);
void (*hpd_enable)(struct drm_bridge *bridge);
void (*hpd_disable)(struct drm_bridge *bridge);
void (*debugfs_init)(struct drm_bridge *bridge, struct dentry *root);
};

```

Members

attach

This callback is invoked whenever our bridge is being attached to a [drm_encoder](#). The flags argument tunes the behaviour of the attach operation (see `DRM_BRIDGE_ATTACH_*`).

The **attach** callback is optional.

RETURNS:

Zero on success, error code on failure.

detach

This callback is invoked whenever our bridge is being detached from a *drm_encoder*.

The **detach** callback is optional.

mode_valid

This callback is used to check if a specific mode is valid in this bridge. This should be implemented if the bridge has some sort of restriction in the modes it can display. For example, a given bridge may be responsible to set a clock value. If the clock can not produce all the values for the available modes then this callback can be used to restrict the number of modes to only the ones that can be displayed.

This hook is used by the probe helpers to filter the mode list in *drm_helper_probe_single_connector_modes()*, and it is used by the atomic helpers to validate modes supplied by userspace in *drm_atomic_helper_check_modeset()*.

The **mode_valid** callback is optional.

NOTE:

Since this function is both called from the check phase of an atomic commit, and the mode validation in the probe paths it is not allowed to look at anything else but the passed-in mode, and validate it against configuration-invariant hardware constraints. Any further limits which depend upon the configuration can only be checked in **mode_fixup**.

RETURNS:

drm_mode_status Enum

mode_fixup

This callback is used to validate and adjust a mode. The parameter mode is the display mode that should be fed to the next element in the display chain, either the final *drm_connector* or the next *drm_bridge*. The parameter adjusted_mode is the input mode the bridge requires. It can be modified by this callback and does not need to match mode. See also *drm_crtc_state.adjusted_mode* for more details.

This is the only hook that allows a bridge to reject a modeset. If this function passes all other callbacks must succeed for this configuration.

The mode_fixup callback is optional. *drm_bridge_funcs.mode_fixup()* is not called when *drm_bridge_funcs.atomic_check()* is implemented, so only one of them should be provided.

NOTE:

This function is called in the check phase of atomic modesets, which can be aborted for any reason (including on userspace's request to just check whether a configuration would be possible). Drivers MUST NOT touch any persistent state (hardware or software) or data structures except the passed in **state** parameter.

Also beware that userspace can request its own custom modes, neither core nor helpers filter modes to the list of probe modes reported by the GETCONNECTOR IOCTL and stored in *drm_connector.modes*. To ensure that modes are filtered consistently put any bridge constraints and limits checks into **mode_valid**.

RETURNS:

True if an acceptable configuration is possible, false if the modeset operation should be rejected.

disable

This callback should disable the bridge. It is called right before the preceding element in the display pipe is disabled. If the preceding element is a bridge this means it's called before that bridge's **disable** vfunc. If the preceding element is a *drm_encoder* it's called right before the *drm_encoder_helper_funcs.disable*, *drm_encoder_helper_funcs.prepare* or *drm_encoder_helper_funcs.dpms* hook.

The bridge can assume that the display pipe (i.e. clocks and timing signals) feeding it is still running when this callback is called.

The **disable** callback is optional.

NOTE:

This is deprecated, do not use! New drivers shall use *drm_bridge_funcs.atomic_disable*.

post_disable

This callback should disable the bridge. It is called right after the preceding element in the display pipe is disabled. If the preceding element is a bridge this means it's called after that bridge's **post_disable** function. If the preceding element is a *drm_encoder* it's called right after the encoder's *drm_encoder_helper_funcs.disable*, *drm_encoder_helper_funcs.prepare* or *drm_encoder_helper_funcs.dpms* hook.

The bridge must assume that the display pipe (i.e. clocks and timing signals) feeding it is no longer running when this callback is called.

The **post_disable** callback is optional.

NOTE:

This is deprecated, do not use! New drivers shall use *drm_bridge_funcs.atomic_post_disable*.

mode_set

This callback should set the given mode on the bridge. It is called after the **mode_set** callback for the preceding element in the display pipeline has been called already. If the bridge is the first element then this would be *drm_encoder_helper_funcs.mode_set*. The display pipe (i.e. clocks and timing signals) is off when this function is called.

The *adjusted_mode* parameter is the mode output by the CRTC for the first bridge in the chain. It can be different from the *mode* parameter that contains the desired mode for the connector at the end of the bridges chain, for instance when the first bridge in the chain performs scaling. The adjusted mode is mostly useful for the first bridge in the chain and is likely irrelevant for the other bridges.

For atomic drivers the *adjusted_mode* is the mode stored in *drm_crtc_state.adjusted_mode*.

NOTE:

This is deprecated, do not use! New drivers shall set their mode in the *drm_bridge_funcs.atomic_enable* operation.

pre_enable

This callback should enable the bridge. It is called right before the preceding element in the display pipe is enabled. If the preceding element is a bridge this means it's called before that bridge's **pre_enable** function. If the preceding element is a *drm_encoder* it's called right before the encoder's *drm_encoder_helper_funcs.enable*, *drm_encoder_helper_funcs.commit* or *drm_encoder_helper_funcs.dpms* hook.

The display pipe (i.e. clocks and timing signals) feeding this bridge will not yet be running when this callback is called. The bridge must not enable the display link feeding the next bridge in the chain (if there is one) when this callback is called.

The **pre_enable** callback is optional.

NOTE:

This is deprecated, do not use! New drivers shall use `drm_bridge_funcs.atomic_pre_enable`.

enable

This callback should enable the bridge. It is called right after the preceding element in the display pipe is enabled. If the preceding element is a bridge this means it's called after that bridge's **enable** function. If the preceding element is a `drm_encoder` it's called right after the encoder's `drm_encoder_helper_funcs.enable`, `drm_encoder_helper_funcs.commit` or `drm_encoder_helper_funcs.dpms` hook.

The bridge can assume that the display pipe (i.e. clocks and timing signals) feeding it is running when this callback is called. This callback must enable the display link feeding the next bridge in the chain if there is one.

The **enable** callback is optional.

NOTE:

This is deprecated, do not use! New drivers shall use `drm_bridge_funcs.atomic_enable`.

atomic_pre_enable

This callback should enable the bridge. It is called right before the preceding element in the display pipe is enabled. If the preceding element is a bridge this means it's called before that bridge's **atomic_pre_enable** or **pre_enable** function. If the preceding element is a `drm_encoder` it's called right before the encoder's `drm_encoder_helper_funcs.atomic_enable` hook.

The display pipe (i.e. clocks and timing signals) feeding this bridge will not yet be running when this callback is called. The bridge must not enable the display link feeding the next bridge in the chain (if there is one) when this callback is called.

The **atomic_pre_enable** callback is optional.

atomic_enable

This callback should enable the bridge. It is called right after the preceding element in the display pipe is enabled. If the preceding element is a bridge this means it's called after that bridge's **atomic_enable** or **enable** function. If the preceding element is a `drm_encoder` it's called right after the encoder's `drm_encoder_helper_funcs.atomic_enable` hook.

The bridge can assume that the display pipe (i.e. clocks and timing signals) feeding it is running when this callback is called. This callback must enable the display link feeding the next bridge in the chain if there is one.

The **atomic_enable** callback is optional.

atomic_disable

This callback should disable the bridge. It is called right before the preceding element in the display pipe is disabled. If the preceding element is a bridge this means it's called before that bridge's **atomic_disable** or **disable** vfunc. If the preceding element is a `drm_encoder` it's called right before the `drm_encoder_helper_funcs.atomic_disable` hook.

The bridge can assume that the display pipe (i.e. clocks and timing signals) feeding it is still running when this callback is called.

The **atomic_disable** callback is optional.

atomic_post_disable

This callback should disable the bridge. It is called right after the preceding element in the display pipe is disabled. If the preceding element is a bridge this means it's called after that bridge's **atomic_post_disable** or **post_disable** function. If the preceding element is a *drm_encoder* it's called right after the encoder's *drm_encoder_helper_funcs.atomic_disable* hook.

The bridge must assume that the display pipe (i.e. clocks and timing signals) feeding it is no longer running when this callback is called.

The **atomic_post_disable** callback is optional.

atomic_duplicate_state

Duplicate the current bridge state object (which is guaranteed to be non-NULL).

The *atomic_duplicate_state* hook is mandatory if the bridge implements any of the atomic hooks, and should be left unassigned otherwise. For bridges that don't subclass *drm_bridge_state*, the *drm_atomic_helper_bridge_duplicate_state()* helper function shall be used to implement this hook.

RETURNS: A valid *drm_bridge_state* object or NULL if the allocation fails.

atomic_destroy_state

Destroy a bridge state object previously allocated by *drm_bridge_funcs.atomic_duplicate_state()*.

The *atomic_destroy_state* hook is mandatory if the bridge implements any of the atomic hooks, and should be left unassigned otherwise. For bridges that don't subclass *drm_bridge_state*, the *drm_atomic_helper_bridge_destroy_state()* helper function shall be used to implement this hook.

atomic_get_output_bus_fmts

Return the supported bus formats on the output end of a bridge. The returned array must be allocated with *kmalloc()* and will be freed by the caller. If the allocation fails, NULL should be returned. *num_output_fmts* must be set to the returned array size. Formats listed in the returned array should be listed in decreasing preference order (the core will try all formats until it finds one that works).

This method is only called on the last element of the bridge chain as part of the bus format negotiation process that happens in *drm_atomic_bridge_chain_select_bus_fmts()*. *This method is optional. When not implemented, the core will fall back to :c:type:`drm_connector.display_info.bus_formats[0]` if *drm_connector.display_info.num_bus_formats* > 0, or to MEDIA_BUS_FMT_FIXED otherwise.*

atomic_get_input_bus_fmts

Return the supported bus formats on the input end of a bridge for a specific output bus format.

The returned array must be allocated with *kmalloc()* and will be freed by the caller. If the allocation fails, NULL should be returned. *num_input_fmts* must be set to the returned array size. Formats listed in the returned array should be listed in decreasing preference order (the core will try all formats until it finds one that works). When the format is not supported NULL should be returned and *num_input_fmts* should be set to 0.

This method is called on all elements of the bridge chain as part of the bus format negotiation process that happens in `drm_atomic_bridge_chain_select_bus_fmts()`. This method is optional. When not implemented, the core will bypass bus format negotiation on this element of the bridge without failing, and the previous element in the chain will be passed `MEDIA_BUS_FMT_FIXED` as its output bus format.

Bridge drivers that need to support being linked to bridges that are not supporting bus format negotiation should handle the `output_fmt == MEDIA_BUS_FMT_FIXED` case appropriately, by selecting a sensible default value or extracting this information from somewhere else (FW property, `drm_display_mode`, `drm_display_info`, ...)

Note: Even if input format selection on the first bridge has no impact on the negotiation process (bus format negotiation stops once we reach the first element of the chain), drivers are expected to return accurate input formats as the input format may be used to configure the CRTC output appropriately.

atomic_check

This method is responsible for checking bridge state correctness. It can also check the state of the surrounding components in chain to make sure the whole pipeline can work properly.

`drm_bridge_funcs.atomic_check()` hooks are called in reverse order (from the last to the first bridge).

This method is optional. `drm_bridge_funcs.mode_fixup()` is not called when `drm_bridge_funcs.atomic_check()` is implemented, so only one of them should be provided.

If drivers need to tweak `drm_bridge_state.input_bus_cfg.flags` or `drm_bridge_state.output_bus_cfg.flags` it should happen in this function. By default the `drm_bridge_state.output_bus_cfg.flags` field is set to the next bridge `drm_bridge_state.input_bus_cfg.flags` value or `drm_connector.display_info.bus_flags` if the bridge is the last element in the chain.

RETURNS: zero if the check passed, a negative error code otherwise.

atomic_reset

Reset the bridge to a predefined state (or retrieve its current state) and return a `drm_bridge_state` object matching this state. This function is called at attach time.

The `atomic_reset` hook is mandatory if the bridge implements any of the atomic hooks, and should be left unassigned otherwise. For bridges that don't subclass `drm_bridge_state`, the `drm_atomic_helper_bridge_reset()` helper function shall be used to implement this hook.

Note that the `atomic_reset()` semantics is not exactly matching the `reset()` semantics found on other components (connector, plane, ...).

1. The reset operation happens when the bridge is attached, not when `drm_mode_config_reset()` is called
2. It's meant to be used exclusively on bridges that have been converted to the ATOMIC API

RETURNS: A valid `drm_bridge_state` object in case of success, an `ERR_PTR()` giving the reason of the failure otherwise.

detect

Check if anything is attached to the bridge output.

This callback is optional, if not implemented the bridge will be considered as always having a component attached to its output. Bridges that implement this callback shall set the `DRM_BRIDGE_OP_DETECT` flag in their `drm_bridge->ops`.

RETURNS:

`drm_connector_status` indicating the bridge output status.

get_modes

Fill all modes currently valid for the sink into the `drm_connector` with `drm_mode_probed_add()`.

The **get_modes** callback is mostly intended to support non-probeable displays such as many fixed panels. Bridges that support reading EDID shall leave **get_modes** unimplemented and implement the `drm_bridge_funcs->get_edid` callback instead.

This callback is optional. Bridges that implement it shall set the `DRM_BRIDGE_OP_MODES` flag in their `drm_bridge->ops`.

The connector parameter shall be used for the sole purpose of filling modes, and shall not be stored internally by bridge drivers for future usage.

RETURNS:

The number of modes added by calling `drm_mode_probed_add()`.

edid_read

Read the EDID data of the connected display.

The **edid_read** callback is the preferred way of reporting mode information for a display connected to the bridge output. Bridges that support reading EDID shall implement this callback and leave the **get_modes** callback unimplemented.

The caller of this operation shall first verify the output connection status and refrain from reading EDID from a disconnected output.

This callback is optional. Bridges that implement it shall set the `DRM_BRIDGE_OP_EDID` flag in their `drm_bridge->ops`.

The connector parameter shall be used for the sole purpose of EDID retrieval, and shall not be stored internally by bridge drivers for future usage.

RETURNS:

An `edid` structure newly allocated with `drm_edid_alloc()` or returned from `drm_edid_read()` family of functions on success, or `NULL` otherwise. The caller is responsible for freeing the returned `edid` structure with `drm_edid_free()`.

get_edid

Read and parse the EDID data of the connected display.

The **get_edid** callback is the preferred way of reporting mode information for a display connected to the bridge output. Bridges that support reading EDID shall implement this callback and leave the **get_modes** callback unimplemented.

The caller of this operation shall first verify the output connection status and refrain from reading EDID from a disconnected output.

This callback is optional. Bridges that implement it shall set the `DRM_BRIDGE_OP_EDID` flag in their `drm_bridge->ops`.

The connector parameter shall be used for the sole purpose of EDID retrieval and parsing, and shall not be stored internally by bridge drivers for future usage.

RETURNS:

An edid structure newly allocated with `kmalloc()` (or similar) on success, or `NULL` otherwise. The caller is responsible for freeing the returned edid structure with `kfree()`.

hpd_notify

Notify the bridge of hot plug detection.

This callback is optional, it may be implemented by bridges that need to be notified of display connection or disconnection for internal reasons. One use case is to reset the internal state of CEC controllers for HDMI bridges.

hpd_enable

Enable hot plug detection. From now on the bridge shall call `drm_bridge_hpd_notify()` each time a change is detected in the output connection status, until hot plug detection gets disabled with **hpd_disable**.

This callback is optional and shall only be implemented by bridges that support hot-plug notification without polling. Bridges that implement it shall also implement the **hpd_disable** callback and set the `DRM_BRIDGE_OP_HPD` flag in their `drm_bridge->ops`.

hpd_disable

Disable hot plug detection. Once this function returns the bridge shall not call `drm_bridge_hpd_notify()` when a change in the output connection status occurs.

This callback is optional and shall only be implemented by bridges that support hot-plug notification without polling. Bridges that implement it shall also implement the **hpd_enable** callback and set the `DRM_BRIDGE_OP_HPD` flag in their `drm_bridge->ops`.

debugfs_init

Allows bridges to create bridge-specific debugfs files.

struct drm_bridge_timings

timing information for the bridge

Definition:

```
struct drm_bridge_timings {
    u32 input_bus_flags;
    u32 setup_time_ps;
    u32 hold_time_ps;
    bool dual_link;
};
```

Members

input_bus_flags

Tells what additional settings for the pixel data on the bus this bridge requires (like pixel signal polarity). See also `drm_display_info->bus_flags`.

setup_time_ps

Defines the time in picoseconds the input data lines must be stable before the clock edge.

hold_time_ps

Defines the time in picoseconds taken for the bridge to sample the input signal after the clock edge.

dual_link

True if the bus operates in dual-link mode. The exact meaning is dependent on the bus type. For LVDS buses, this indicates that even- and odd-numbered pixels are received on separate links.

enum drm_bridge_ops

Bitmask of operations supported by the bridge

Constants**DRM_BRIDGE_OP_DETECT**

The bridge can detect displays connected to its output. Bridges that set this flag shall implement the *drm_bridge_funcs->detect* callback.

DRM_BRIDGE_OP_EDID

The bridge can retrieve the EDID of the display connected to its output. Bridges that set this flag shall implement the *drm_bridge_funcs->get_edid* callback.

DRM_BRIDGE_OP_HPD

The bridge can detect hot-plug and hot-unplug without requiring polling. Bridges that set this flag shall implement the *drm_bridge_funcs->hpd_enable* and *drm_bridge_funcs->hpd_disable* callbacks if they support enabling and disabling hot-plug detection dynamically.

DRM_BRIDGE_OP_MODES

The bridge can retrieve the modes supported by the display at its output. This does not include reading EDID which is separately covered by **DRM_BRIDGE_OP_EDID**. Bridges that set this flag shall implement the *drm_bridge_funcs->get_modes* callback.

struct drm_bridge

central DRM bridge control structure

Definition:

```
struct drm_bridge {
    struct drm_private_obj base;
    struct drm_device *dev;
    struct drm_encoder *encoder;
    struct list_head chain_node;
#ifdef CONFIG_OF;
    struct device_node *of_node;
#endif;
    struct list_head list;
    const struct drm_bridge_timings *timings;
    const struct drm_bridge_funcs *funcs;
    void *driver_private;
    enum drm_bridge_ops ops;
    int type;
    bool interlace_allowed;
    bool pre_enable_prev_first;
    struct i2c_adapter *ddc;
```

```
struct mutex hpd_mutex;
void (*hpd_cb)(void *data, enum drm_connector_status status);
void *hpd_data;
};
```

Members

base

inherit from `drm_private_object`

dev

DRM device this bridge belongs to

encoder

encoder to which this bridge is connected

chain_node

used to form a bridge chain

of_node

device node pointer to the bridge

list

to keep track of all added bridges

timings

the timing specification for the bridge, if any (may be NULL)

funcs

control functions

driver_private

pointer to the bridge driver's internal context

ops

bitmask of operations supported by the bridge

type

Type of the connection at the bridge output (`DRM_MODE_CONNECTOR_*`). For bridges at the end of this chain this identifies the type of connected display.

interlace_allowed

Indicate that the bridge can handle interlaced modes.

pre_enable_prev_first

The bridge requires that the prev bridge **pre_enable** function is called before its **pre_enable**, and conversely for `post_disable`. This is most frequently a requirement for DSI devices which need the host to be initialised before the peripheral.

ddc

Associated I2C adapter for DDC access, if any.

hpd_mutex

Protects the **hpd_cb** and **hpd_data** fields.

hpd_cb

Hot plug detection callback, registered with `drm_bridge_hpd_enable()`.

hpd_data

Private data passed to the Hot plug detection callback **hpd_cb**.

struct *drm_bridge* ***drm_bridge_get_next_bridge**(struct *drm_bridge* *bridge)

Get the next bridge in the chain

Parameters

struct *drm_bridge* ***bridge**

bridge object

Return

the next bridge in the chain after **bridge**, or NULL if **bridge** is the last.

struct *drm_bridge* ***drm_bridge_get_prev_bridge**(struct *drm_bridge* *bridge)

Get the previous bridge in the chain

Parameters

struct *drm_bridge* ***bridge**

bridge object

Return

the previous bridge in the chain, or NULL if **bridge** is the first.

struct *drm_bridge* ***drm_bridge_chain_get_first_bridge**(struct *drm_encoder* *encoder)

Get the first bridge in the chain

Parameters

struct *drm_encoder* ***encoder**

encoder object

Return

the first bridge in the chain, or NULL if **encoder** has no bridge attached to it.

drm_for_each_bridge_in_chain

drm_for_each_bridge_in_chain (encoder, bridge)

Iterate over all bridges present in a chain

Parameters

encoder

the encoder to iterate bridges on

bridge

a bridge pointer updated to point to the current bridge at each iteration

Description

Iterate over all bridges present in the bridge chain attached to **encoder**.

void **drm_bridge_add**(struct *drm_bridge* *bridge)

add the given bridge to the global bridge list

Parameters

struct drm_bridge *bridge

bridge control structure

int **devm_drm_bridge_add**(struct device *dev, struct *drm_bridge* *bridge)

devm managed version of *drm_bridge_add()*

Parameters

struct device *dev

device to tie the bridge lifetime to

struct drm_bridge *bridge

bridge control structure

Description

This is the managed version of *drm_bridge_add()* which automatically calls *drm_bridge_remove()* when **dev** is unbound.

Return

0 if no error or negative error code.

void **drm_bridge_remove**(struct *drm_bridge* *bridge)

remove the given bridge from the global bridge list

Parameters

struct drm_bridge *bridge

bridge control structure

int **drm_bridge_attach**(struct *drm_encoder* *encoder, struct *drm_bridge* *bridge, struct *drm_bridge* *previous, enum *drm_bridge_attach_flags* flags)

attach the bridge to an encoder's chain

Parameters

struct drm_encoder *encoder

DRM encoder

struct drm_bridge *bridge

bridge to attach

struct drm_bridge *previous

previous bridge in the chain (optional)

enum drm_bridge_attach_flags flags

DRM_BRIDGE_ATTACH_* flags

Description

Called by a kms driver to link the bridge to an encoder's chain. The previous argument specifies the previous bridge in the chain. If NULL, the bridge is linked directly at the encoder's output. Otherwise it is linked at the previous bridge's output.

If non-NULL the previous bridge must be already attached by a call to this function.

Note that bridges attached to encoders are auto-detached during encoder cleanup in *drm_encoder_cleanup()*, so *drm_bridge_attach()* should generally *not* be balanced with a *drm_bridge_detach()* in driver code.

Return

Zero on success, error code on failure

```
bool drm_bridge_chain_mode_fixup(struct drm_bridge *bridge, const struct  
                                drm_display_mode *mode, struct drm_display_mode  
                                *adjusted_mode)
```

fixup proposed mode for all bridges in the encoder chain

Parameters

struct *drm_bridge* *bridge
bridge control structure

const struct *drm_display_mode* *mode
desired mode to be set for the bridge

struct *drm_display_mode* *adjusted_mode
updated mode that works for this bridge

Description

Calls *drm_bridge_funcs.mode_fixup* for all the bridges in the encoder chain, starting from the first bridge to the last.

Note

the bridge passed should be the one closest to the encoder

Return

true on success, false on failure

```
enum drm_mode_status drm_bridge_chain_mode_valid(struct drm_bridge *bridge, const  
                                                  struct drm_display_info *info, const  
                                                  struct drm_display_mode *mode)
```

validate the mode against all bridges in the encoder chain.

Parameters

struct *drm_bridge* *bridge
bridge control structure

const struct *drm_display_info* *info
display info against which the mode shall be validated

const struct *drm_display_mode* *mode
desired mode to be validated

Description

Calls *drm_bridge_funcs.mode_valid* for all the bridges in the encoder chain, starting from the first bridge to the last. If at least one bridge does not accept the mode the function returns the error code.

Note

the bridge passed should be the one closest to the encoder.

Return

MODE_OK on success, *drm_mode_status* Enum error code on failure

void **drm_bridge_chain_mode_set**(struct *drm_bridge* *bridge, const struct *drm_display_mode* *mode, const struct *drm_display_mode* *adjusted_mode)

set proposed mode for all bridges in the encoder chain

Parameters

struct drm_bridge *bridge

bridge control structure

const struct drm_display_mode *mode

desired mode to be set for the encoder chain

const struct drm_display_mode *adjusted_mode

updated mode that works for this encoder chain

Description

Calls *drm_bridge_funcs.mode_set* op for all the bridges in the encoder chain, starting from the first bridge to the last.

Note

the bridge passed should be the one closest to the encoder

void **drm_atomic_bridge_chain_disable**(struct *drm_bridge* *bridge, struct *drm_atomic_state* *old_state)

disables all bridges in the encoder chain

Parameters

struct drm_bridge *bridge

bridge control structure

struct drm_atomic_state *old_state

old atomic state

Description

Calls *drm_bridge_funcs.atomic_disable* (falls back on *drm_bridge_funcs.disable*) op for all the bridges in the encoder chain, starting from the last bridge to the first. These are called before calling *drm_encoder_helper_funcs.atomic_disable*

Note

the bridge passed should be the one closest to the encoder

void **drm_atomic_bridge_chain_post_disable**(struct *drm_bridge* *bridge, struct *drm_atomic_state* *old_state)

cleans up after disabling all bridges in the encoder chain

Parameters

struct drm_bridge *bridge

bridge control structure

struct drm_atomic_state *old_state

old atomic state

Description

Calls `drm_bridge_funcs.atomic_post_disable` (falls back on `drm_bridge_funcs.post_disable`) op for all the bridges in the encoder chain, starting from the first bridge to the last. These are called after completing `drm_encoder_helper_funcs.atomic_disable`

If a bridge sets **pre_enable_prev_first**, then the **post_disable** for that bridge will be called before the previous one to reverse the **pre_enable** calling direction.

Note

the bridge passed should be the one closest to the encoder

```
void drm_atomic_bridge_chain_pre_enable(struct drm_bridge *bridge, struct drm_atomic_state *old_state)
```

prepares for enabling all bridges in the encoder chain

Parameters

struct *drm_bridge* *bridge
bridge control structure

struct *drm_atomic_state* *old_state
old atomic state

Description

Calls `drm_bridge_funcs.atomic_pre_enable` (falls back on `drm_bridge_funcs.pre_enable`) op for all the bridges in the encoder chain, starting from the last bridge to the first. These are called before calling `drm_encoder_helper_funcs.atomic_enable`

If a bridge sets **pre_enable_prev_first**, then the `pre_enable` for the prev bridge will be called before `pre_enable` of this bridge.

Note

the bridge passed should be the one closest to the encoder

```
void drm_atomic_bridge_chain_enable(struct drm_bridge *bridge, struct drm_atomic_state *old_state)
```

enables all bridges in the encoder chain

Parameters

struct *drm_bridge* *bridge
bridge control structure

struct *drm_atomic_state* *old_state
old atomic state

Description

Calls `drm_bridge_funcs.atomic_enable` (falls back on `drm_bridge_funcs.enable`) op for all the bridges in the encoder chain, starting from the first bridge to the last. These are called after completing `drm_encoder_helper_funcs.atomic_enable`

Note

the bridge passed should be the one closest to the encoder

```
int drm_atomic_bridge_chain_check(struct drm_bridge *bridge, struct drm_crtc_state *crtc_state, struct drm_connector_state *conn_state)
```

Do an atomic check on the bridge chain

Parameters

struct drm_bridge *bridge
bridge control structure

struct drm_crtc_state *crtc_state
new CRTC state

struct drm_connector_state *conn_state
new connector state

Description

First trigger a bus format negotiation before calling `drm_bridge_funcs.atomic_check()` (falls back on `drm_bridge_funcs.mode_fixup()`) op for all the bridges in the encoder chain, starting from the last bridge to the first. These are called before calling `drm_encoder_helper_funcs.atomic_check()`

Return

0 on success, a negative error code on failure

enum `drm_connector_status` **drm_bridge_detect**(struct `drm_bridge` *bridge)
check if anything is attached to the bridge output

Parameters

struct drm_bridge *bridge
bridge control structure

Description

If the bridge supports output detection, as reported by the `DRM_BRIDGE_OP_DETECT` bridge ops flag, call `drm_bridge_funcs.detect` for the bridge and return the connection status. Otherwise return `connector_status_unknown`.

Return

The detection status on success, or `connector_status_unknown` if the bridge doesn't support output detection.

int **drm_bridge_get_modes**(struct `drm_bridge` *bridge, struct `drm_connector` *connector)
fill all modes currently valid for the sink into the **connector**

Parameters

struct drm_bridge *bridge
bridge control structure

struct drm_connector *connector
the connector to fill with modes

Description

If the bridge supports output modes retrieval, as reported by the `DRM_BRIDGE_OP_MODES` bridge ops flag, call `drm_bridge_funcs.get_modes` to fill the connector with all valid modes and return the number of modes added. Otherwise return 0.

Return

The number of modes added to the connector.


```
const struct drm_edid *drm_bridge_edid_read(struct drm_bridge *bridge, struct  
                                             drm_connector *connector)
```

read the EDID data of the connected display

Parameters

struct drm_bridge *bridge
bridge control structure

struct drm_connector *connector
the connector to read EDID for

Description

If the bridge supports output EDID retrieval, as reported by the `DRM_BRIDGE_OP_EDID` bridge ops flag, call *drm_bridge_funcs.edid_read* to get the EDID and return it. Otherwise return `NULL`.

If *drm_bridge_funcs.edid_read* is not set, fall back to using *drm_bridge_get_edid()* and wrapping it in struct `drm_edid`.

Return

The retrieved EDID on success, or `NULL` otherwise.

```
struct edid *drm_bridge_get_edid(struct drm_bridge *bridge, struct drm_connector  
                                  *connector)
```

get the EDID data of the connected display

Parameters

struct drm_bridge *bridge
bridge control structure

struct drm_connector *connector
the connector to read EDID for

Description

If the bridge supports output EDID retrieval, as reported by the `DRM_BRIDGE_OP_EDID` bridge ops flag, call *drm_bridge_funcs.get_edid* to get the EDID and return it. Otherwise return `NULL`.

Deprecated. Prefer using *drm_bridge_edid_read()*.

Return

The retrieved EDID on success, or `NULL` otherwise.

```
void drm_bridge_hpd_enable(struct drm_bridge *bridge, void (*cb)(void *data, enum  
                                                                drm_connector_status status), void *data)
```

enable hot plug detection for the bridge

Parameters

struct drm_bridge *bridge
bridge control structure

void (*cb)(void *data, enum drm_connector_status status)
hot-plug detection callback

void *data

data to be passed to the hot-plug detection callback

Description

Call *drm_bridge_funcs.hpd_enable* if implemented and register the given **cb** and **data** as hot plug notification callback. From now on the **cb** will be called with **data** when an output status change is detected by the bridge, until hot plug notification gets disabled with *drm_bridge_hpd_disable()*.

Hot plug detection is supported only if the `DRM_BRIDGE_OP_HPD` flag is set in `bridge->ops`. This function shall not be called when the flag is not set.

Only one hot plug detection callback can be registered at a time, it is an error to call this function when hot plug detection is already enabled for the bridge.

void **drm_bridge_hpd_disable**(struct *drm_bridge* *bridge)

disable hot plug detection for the bridge

Parameters

struct **drm_bridge** *bridge

bridge control structure

Description

Call *drm_bridge_funcs.hpd_disable* if implemented and unregister the hot plug detection callback previously registered with *drm_bridge_hpd_enable()*. Once this function returns the callback will not be called by the bridge when an output status change occurs.

Hot plug detection is supported only if the `DRM_BRIDGE_OP_HPD` flag is set in `bridge->ops`. This function shall not be called when the flag is not set.

void **drm_bridge_hpd_notify**(struct *drm_bridge* *bridge, enum *drm_connector_status* status)

notify hot plug detection events

Parameters

struct **drm_bridge** *bridge

bridge control structure

enum **drm_connector_status** status

output connection status

Description

Bridge drivers shall call this function to report hot plug events when they detect a change in the output status, when hot plug detection has been enabled by *drm_bridge_hpd_enable()*.

This function shall be called in a context that can sleep.

struct *drm_bridge* ***of_drm_find_bridge**(struct device_node *np)

find the bridge corresponding to the device node in the global bridge list

Parameters

struct **device_node** *np

device node

Return

drm_bridge control struct on success, NULL on failure

5.8.7 MIPI-DSI bridge operation

DSI host interfaces are expected to be implemented as bridges rather than encoders, however there are a few aspects of their operation that need to be defined in order to provide a consistent interface.

A DSI host should keep the PHY powered down until the `pre_enable` operation is called. All lanes are in an undefined idle state up to this point, and it must not be assumed that it is LP-11. `pre_enable` should initialise the PHY, set the data lanes to LP-11, and the clock lane to either LP-11 or HS depending on the mode_flag `MIPI_DSI_CLOCK_NON_CONTINUOUS`.

Ordinarily the downstream bridge DSI peripheral `pre_enable` will have been called before the DSI host. If the DSI peripheral requires LP-11 and/or the clock lane to be in HS mode prior to `pre_enable`, then it can set the `pre_enable_prev_first` flag to request the `pre_enable` (and `post_disable`) order to be altered to enable the DSI host first.

Either the CRTC being enabled, or the DSI host enable operation should switch the host to actively transmitting video on the data lanes.

The reverse also applies. The DSI host disable operation or stopping the CRTC should stop transmitting video, and the data lanes should return to the LP-11 state. The DSI host `post_disable` operation should disable the PHY. If the `pre_enable_prev_first` flag is set, then the DSI peripheral's bridge `post_disable` will be called before the DSI host's `post_disable`.

Whilst it is valid to call `host_transfer` prior to `pre_enable` or after `post_disable`, the exact state of the lanes is undefined at this point. The DSI host should initialise the interface, transmit the data, and then disable the interface again.

Ultra Low Power State (ULPS) is not explicitly supported by DRM. If implemented, it therefore needs to be handled entirely within the DSI Host driver.

5.8.8 Bridge Connector Helper Reference

```
struct drm_connector *drm_bridge_connector_init(struct drm_device *drm, struct
                                              drm_encoder *encoder)
```

Initialise a connector for a chain of bridges

Parameters

struct *drm_device* *drm
the DRM device

struct *drm_encoder* *encoder
the encoder where the bridge chain starts

Description

Allocate, initialise and register a `drm_bridge_connector` with the **drm** device. The connector is associated with a chain of bridges that starts at the **encoder**. All bridges in the chain shall report bridge operation flags (`drm_bridge->ops`) and bridge output type (`drm_bridge->type`), and none of them may create a DRM connector directly.

Returns a pointer to the new connector on success, or a negative error pointer otherwise.

5.8.9 Panel-Bridge Helper Reference

bool **drm_bridge_is_panel**(const struct *drm_bridge* *bridge)

Checks if a *drm_bridge* is a *panel_bridge*.

Parameters

const struct *drm_bridge* *bridge

The *drm_bridge* to be checked.

Description

Returns true if the bridge is a panel bridge, or false otherwise.

struct *drm_bridge* ***drm_panel_bridge_add**(struct *drm_panel* *panel)

Creates a *drm_bridge* and *drm_connector* that just calls the appropriate functions from *drm_panel*.

Parameters

struct *drm_panel* *panel

The *drm_panel* being wrapped. Must be non-NULL.

Description

For drivers converting from directly using *drm_panel*: The expected usage pattern is that during either encoder module probe or DSI host attach, a *drm_panel* will be looked up through *drm_of_find_panel_or_bridge()*. *drm_panel_bridge_add()* is used to wrap that panel in the new bridge, and the result can then be passed to *drm_bridge_attach()*. The *drm_panel_prepare()* and related functions can be dropped from the encoder driver (they're now called by the KMS helpers before calling into the encoder), along with connector creation. When done with the bridge (after *drm_mode_config_cleanup()* if the bridge has already been attached), then *drm_panel_bridge_remove()* to free it.

The connector type is set to **panel->connector_type**, which must be set to a known type. Calling this function with a panel whose connector type is *DRM_MODE_CONNECTOR_Unknown* will return *ERR_PTR(-EINVAL)*.

See *devm_drm_panel_bridge_add()* for an automatically managed version of this function.

struct *drm_bridge* ***drm_panel_bridge_add_typed**(struct *drm_panel* *panel, u32
connector_type)

Creates a *drm_bridge* and *drm_connector* with an explicit connector type.

Parameters

struct *drm_panel* *panel

The *drm_panel* being wrapped. Must be non-NULL.

u32 connector_type

The connector type (*DRM_MODE_CONNECTOR_**)

Description

This is just like *drm_panel_bridge_add()*, but forces the connector type to **connector_type** instead of inferring it from the panel.

This function is deprecated and should not be used in new drivers. Use *drm_panel_bridge_add()* instead, and fix panel drivers as necessary if they don't report a connector type.

void **drm_panel_bridge_remove**(struct *drm_bridge* *bridge)

Unregisters and frees a *drm_bridge* created by *drm_panel_bridge_add()*.

Parameters

struct *drm_bridge* *bridge

The *drm_bridge* being freed.

int **drm_panel_bridge_set_orientation**(struct *drm_connector* *connector, struct *drm_bridge* *bridge)

Set the connector's panel orientation from the bridge that can be transformed to panel bridge.

Parameters

struct *drm_connector* *connector

The connector to be set panel orientation.

struct *drm_bridge* *bridge

The *drm_bridge* to be transformed to panel bridge.

Description

Returns 0 on success, negative *errno* on failure.

struct *drm_bridge* ***devm_drm_panel_bridge_add**(struct device *dev, struct *drm_panel* *panel)

Creates a managed *drm_bridge* and *drm_connector* that just calls the appropriate functions from *drm_panel*.

Parameters

struct device *dev

device to tie the bridge lifetime to

struct *drm_panel* *panel

The *drm_panel* being wrapped. Must be non-NULL.

Description

This is the managed version of *drm_panel_bridge_add()* which automatically calls *drm_panel_bridge_remove()* when **dev** is unbound.

struct *drm_bridge* ***devm_drm_panel_bridge_add_typed**(struct device *dev, struct *drm_panel* *panel, u32 connector_type)

Creates a managed *drm_bridge* and *drm_connector* with an explicit connector type.

Parameters

struct device *dev

device to tie the bridge lifetime to

struct *drm_panel* *panel

The *drm_panel* being wrapped. Must be non-NULL.

u32 connector_type

The connector type (DRM_MODE_CONNECTOR_*)

Description

This is just like `devm_drm_panel_bridge_add()`, but forces the connector type to **connector_type** instead of inferring it from the panel.

This function is deprecated and should not be used in new drivers. Use `devm_drm_panel_bridge_add()` instead, and fix panel drivers as necessary if they don't report a connector type.

```
struct drm_bridge *drmm_panel_bridge_add(struct drm_device *drm, struct drm_panel
                                         *panel)
```

Creates a DRM-managed *drm_bridge* and *drm_connector* that just calls the appropriate functions from *drm_panel*.

Parameters

struct *drm_device* *drm

DRM device to tie the bridge lifetime to

struct *drm_panel* *panel

The *drm_panel* being wrapped. Must be non-NULL.

Description

This is the DRM-managed version of `drm_panel_bridge_add()` which automatically calls `drm_panel_bridge_remove()` when **dev** is cleaned up.

```
struct drm_connector *drm_panel_bridge_connector(struct drm_bridge *bridge)
    return the connector for the panel bridge
```

Parameters

struct *drm_bridge* *bridge

The *drm_bridge*.

Description

`drm_panel_bridge` creates the connector. This function gives external access to the connector.

Return

Pointer to *drm_connector*

```
struct drm_bridge *devm_drm_of_get_bridge(struct device *dev, struct device_node *np, u32
                                         port, u32 endpoint)
```

Return next bridge in the chain

Parameters

struct *device* *dev

device to tie the bridge lifetime to

struct *device_node* *np

device tree node containing encoder output ports

u32 port

port in the device tree node

u32 endpoint

endpoint in the device tree node

Description

Given a DT node's port and endpoint number, finds the connected node and returns the associated bridge if any, or creates and returns a drm panel bridge instance if a panel is connected.

Returns a pointer to the bridge if successful, or an error pointer otherwise.

```
struct drm_bridge *drmm_of_get_bridge(struct drm_device *drm, struct device_node *np,
                                     u32 port, u32 endpoint)
```

Return next bridge in the chain

Parameters

struct *drm_device* *drm
device to tie the bridge lifetime to

struct *device_node* *np
device tree node containing encoder output ports

u32 port
port in the device tree node

u32 endpoint
endpoint in the device tree node

Description

Given a DT node's port and endpoint number, finds the connected node and returns the associated bridge if any, or creates and returns a drm panel bridge instance if a panel is connected.

Returns a drmm managed pointer to the bridge if successful, or an error pointer otherwise.

5.9 Panel Helper Reference

The DRM panel helpers allow drivers to register panel objects with a central registry and provide functions to retrieve those panels in display drivers.

For easy integration into drivers using the *drm_bridge* infrastructure please take look at *drm_panel_bridge_add()* and *devm_drm_panel_bridge_add()*.

struct **drm_panel_funcs**
perform operations on a given panel

Definition:

```
struct drm_panel_funcs {
    int (*prepare)(struct drm_panel *panel);
    int (*enable)(struct drm_panel *panel);
    int (*disable)(struct drm_panel *panel);
    int (*unprepare)(struct drm_panel *panel);
    int (*get_modes)(struct drm_panel *panel, struct drm_connector *connector);
    enum drm_panel_orientation (*get_orientation)(struct drm_panel *panel);
    int (*get_timings)(struct drm_panel *panel, unsigned int num_timings,
    ↪ struct display_timing *timings);
    void (*debugfs_init)(struct drm_panel *panel, struct dentry *root);
};
```

Members

prepare

Turn on panel and perform set up.

This function is optional.

enable

Enable panel (turn on back light, etc.).

This function is optional.

disable

Disable panel (turn off back light, etc.).

This function is optional.

unprepare

Turn off panel.

This function is optional.

get_modes

Add modes to the connector that the panel is attached to and returns the number of modes added.

This function is mandatory.

get_orientation

Return the panel orientation set by device tree or EDID.

This function is optional.

get_timings

Copy display timings into the provided array and return the number of display timings available.

This function is optional.

debugfs_init

Allows panels to create panels-specific debugfs files.

Description

The `.prepare()` function is typically called before the display controller starts to transmit video data. Panel drivers can use this to turn the panel on and wait for it to become ready. If additional configuration is required (via a control bus such as I2C, SPI or DSI for example) this is a good time to do that.

After the display controller has started transmitting video data, it's safe to call the `.enable()` function. This will typically enable the backlight to make the image on screen visible. Some panels require a certain amount of time or frames before the image is displayed. This function is responsible for taking this into account before enabling the backlight to avoid visual glitches.

Before stopping video transmission from the display controller it can be necessary to turn off the panel to avoid visual glitches. This is done in the `.disable()` function. Analogously to `.enable()` this typically involves turning off the backlight and waiting for some time to make sure no image is visible on the panel. It is then safe for the display controller to cease transmission of video data.

To save power when no video data is transmitted, a driver can power down the panel. This is the job of the `.unprepare()` function.

Backlight can be handled automatically if configured using [`drm_panel_of_backlight\(\)`](#) or [`drm_panel_dp_aux_backlight\(\)`](#). Then the driver does not need to implement the functionality to enable/disable backlight.

struct **drm_panel**

DRM panel object

Definition:

```
struct drm_panel {
    struct device *dev;
    struct backlight_device *backlight;
    const struct drm_panel_funcs *funcs;
    int connector_type;
    struct list_head list;
    struct list_head followers;
    struct mutex follower_lock;
    bool prepare_prev_first;
    bool prepared;
    bool enabled;
};
```

Members

dev

Parent device of the panel.

backlight

Backlight device, used to turn on backlight after the call to `enable()`, and to turn off backlight before the call to `disable()`. `backlight` is set by [`drm_panel_of_backlight\(\)`](#) or [`drm_panel_dp_aux_backlight\(\)`](#) and drivers shall not assign it.

funcs

Operations that can be performed on the panel.

connector_type

Type of the panel as a `DRM_MODE_CONNECTOR_*` value. This is used to initialise the `drm_connector` corresponding to the panel with the correct connector type.

list

Panel entry in registry.

followers

A list of struct `drm_panel_follower` dependent on this panel.

follower_lock

Lock for followers list.

prepare_prev_first

The previous controller should be prepared first, before the prepare for the panel is called. This is largely required for DSI panels where the DSI host controller should be initialised to LP-11 before the panel is powered up.

prepared

If true then the panel has been prepared.

enabled

If true then the panel has been enabled.

```
void drm_panel_init(struct drm_panel *panel, struct device *dev, const struct  
                   drm_panel_funcs *funcs, int connector_type)
```

initialize a panel

Parameters

struct drm_panel *panel
DRM panel

struct device *dev
parent device of the panel

const struct drm_panel_funcs *funcs
panel operations

int connector_type
the connector type (DRM_MODE_CONNECTOR_*) corresponding to the panel interface

Description

Initialize the panel structure for subsequent registration with *drm_panel_add()*.

```
void drm_panel_add(struct drm_panel *panel)  
    add a panel to the global registry
```

Parameters

struct drm_panel *panel
panel to add

Description

Add a panel to the global registry so that it can be looked up by display drivers.

```
void drm_panel_remove(struct drm_panel *panel)  
    remove a panel from the global registry
```

Parameters

struct drm_panel *panel
DRM panel

Description

Removes a panel from the global registry.

```
int drm_panel_prepare(struct drm_panel *panel)  
    power on a panel
```

Parameters

struct drm_panel *panel
DRM panel

Description

Calling this function will enable power and deassert any reset signals to the panel. After this has completed it is possible to communicate with any integrated circuitry via a command bus.

Return

0 on success or a negative error code on failure.

int **drm_panel_unprepare**(struct *drm_panel* *panel)
power off a panel

Parameters

struct drm_panel *panel
DRM panel

Description

Calling this function will completely power off a panel (assert the panel's reset, turn off power supplies, ...). After this function has completed, it is usually no longer possible to communicate with the panel until another call to *drm_panel_prepare()*.

Return

0 on success or a negative error code on failure.

int **drm_panel_enable**(struct *drm_panel* *panel)
enable a panel

Parameters

struct drm_panel *panel
DRM panel

Description

Calling this function will cause the panel display drivers to be turned on and the backlight to be enabled. Content will be visible on screen after this call completes.

Return

0 on success or a negative error code on failure.

int **drm_panel_disable**(struct *drm_panel* *panel)
disable a panel

Parameters

struct drm_panel *panel
DRM panel

Description

This will typically turn off the panel's backlight or disable the display drivers. For smart panels it should still be possible to communicate with the integrated circuitry via any command bus after this call.

Return

0 on success or a negative error code on failure.

int **drm_panel_get_modes**(struct *drm_panel* *panel, struct *drm_connector* *connector)
probe the available display modes of a panel

Parameters

struct drm_panel *panel

DRM panel

struct drm_connector *connector

DRM connector

Description

The modes probed from the panel are automatically added to the connector that the panel is attached to.

Return

The number of modes available from the panel on success, or 0 on failure (no modes).

struct *drm_panel* ***of_drm_find_panel**(const struct device_node *np)

look up a panel using a device tree node

Parameters

const struct device_node *np

device tree node of the panel

Description

Searches the set of registered panels for one that matches the given device tree node. If a matching panel is found, return a pointer to it.

Possible error codes returned by this function:

- EPROBE_DEFER: the panel device has not been probed yet, and the caller should retry later
- ENODEV: the device is not available (status != “okay” or “ok”)

Return

A pointer to the panel registered for the specified device tree node or an ERR_PTR() if no panel matching the device tree node can be found.

int **of_drm_get_panel_orientation**(const struct device_node *np, enum *drm_panel_orientation* *orientation)

look up the orientation of the panel through the “rotation” binding from a device tree node

Parameters

const struct device_node *np

device tree node of the panel

enum drm_panel_orientation *orientation

orientation enum to be filled in

Description

Looks up the rotation of a panel in the device tree. The orientation of the panel is expressed as a property name “rotation” in the device tree. The rotation in the device tree is counter clockwise.

Return

0 when a valid rotation value (0, 90, 180, or 270) is read or the rotation property doesn’t exist. Return a negative error code on failure.

bool **drm_is_panel_follower**(struct device *dev)

Check if the device is a panel follower

Parameters

struct device *dev

The 'struct device' to check

Description

This checks to see if a device needs to be power sequenced together with a panel using the panel follower API. At the moment panels can only be followed on device tree enabled systems. The "panel" property of the follower points to the panel to be followed.

Return

true if we should be power sequenced with a panel; false otherwise.

int **drm_panel_add_follower**(struct device *follower_dev, struct drm_panel_follower *follower)

Register something to follow panel state.

Parameters

struct device *follower_dev

The 'struct device' for the follower.

struct drm_panel_follower *follower

The panel follower descriptor for the follower.

Description

A panel follower is called right after preparing the panel and right before unpreparing the panel. It's primary intention is to power on an associated touchscreen, though it could be used for any similar devices. Multiple devices are allowed the follow the same panel.

If a follower is added to a panel that's already been turned on, the follower's prepare callback is called right away.

At the moment panels can only be followed on device tree enabled systems. The "panel" property of the follower points to the panel to be followed.

Return

0 or an error code. Note that -ENODEV means that we detected that

follower_dev is not actually following a panel. The caller may choose to ignore this return value if following a panel is optional.

void **drm_panel_remove_follower**(struct drm_panel_follower *follower)

Reverse [drm_panel_add_follower\(\)](#).

Parameters

struct drm_panel_follower *follower

The panel follower descriptor for the follower.

Description

Undo [drm_panel_add_follower\(\)](#). This includes calling the follower's unprepare function if we're removed from a panel that's currently prepared.

Return

0 or an error code.

int **devm_drm_panel_add_follower**(struct device *follower_dev, struct drm_panel_follower *follower)

devm version of [drm_panel_add_follower\(\)](#)

Parameters

struct device *follower_dev

The 'struct device' for the follower.

struct drm_panel_follower *follower

The panel follower descriptor for the follower.

Description

Handles calling [drm_panel_remove_follower\(\)](#) using devm on the follower_dev.

Return

0 or an error code.

int **drm_panel_of_backlight**(struct [drm_panel](#) *panel)

use backlight device node for backlight

Parameters

struct drm_panel *panel

DRM panel

Description

Use this function to enable backlight handling if your panel uses device tree and has a backlight phandle.

When the panel is enabled backlight will be enabled after a successful call to [drm_panel_funcs.enable\(\)](#)

When the panel is disabled backlight will be disabled before the call to [drm_panel_funcs.disable\(\)](#).

A typical implementation for a panel driver supporting device tree will call this function at probe time. Backlight will then be handled transparently without requiring any intervention from the driver. [drm_panel_of_backlight\(\)](#) must be called after the call to [drm_panel_init\(\)](#).

Return

0 on success or a negative error code on failure.

int **drm_get_panel_orientation_quirk**(int width, int height)

Check for panel orientation quirks

Parameters

int width

width in pixels of the panel

int height

height in pixels of the panel

Description

This function checks for platform specific (e.g. DMI based) quirks providing info on `panel_orientation` for systems where this cannot be probed from the hard-/firm-ware. To avoid false-positive this function takes the panel resolution as argument and checks that against the resolution expected by the quirk-table entry.

Note this function is also used outside of the `drm`-subsys, by for example the `efifb` code. Because of this this function gets compiled into its own kernel-module when built as a module.

Return

A `DRM_MODE_PANEL_ORIENTATION_*` value if there is a quirk for this system, or `DRM_MODE_PANEL_ORIENTATION_UNKNOWN` if there is no quirk.

5.10 Panel Self Refresh Helper Reference

This helper library provides an easy way for drivers to leverage the atomic framework to implement panel self refresh (SR) support. Drivers are responsible for initializing and cleaning up the SR helpers on load/unload (see [`drm_self_refresh_helper_init/drm_self_refresh_helper_cleanup`](#)). The connector is responsible for setting `drm_connector_state.self_refresh_aware` to true at runtime if it is SR-aware (meaning it knows how to initiate self refresh on the panel).

Once a `crtc` has enabled SR using [`drm_self_refresh_helper_init`](#), the helpers will monitor activity and call back into the driver to enable/disable SR as appropriate. The best way to think about this is that it's a DPMS on/off request with `drm_crtc_state.self_refresh_active` set in `crtc` state that tells you to disable/enable SR on the panel instead of power-cycling it.

During SR, drivers may choose to fully disable their `crtc`/encoder/bridge hardware (in which case no driver changes are necessary), or they can inspect `drm_crtc_state.self_refresh_active` if they want to enter low power mode without full disable (in case full disable/enable is too slow).

SR will be deactivated if there are any atomic updates affecting the pipe that is in SR mode. If a `crtc` is driving multiple connectors, all connectors must be SR aware and all will enter/exit SR mode at the same time.

If the `crtc` and connector are SR aware, but the panel connected does not support it (or is otherwise unable to enter SR), the driver should fail `atomic_check` when `drm_crtc_state.self_refresh_active` is true.

```
void drm_self_refresh_helper_update_avg_times(struct drm\_atomic\_state *state,  
                                              unsigned int commit_time_ms, unsigned  
                                              int new_self_refresh_mask)
```

Updates a `crtc`'s SR time averages

Parameters

struct `drm_atomic_state` *state
the state which has just been applied to hardware

unsigned int `commit_time_ms`
the amount of time in ms that this commit took to complete

unsigned int `new_self_refresh_mask`
bitmask of `crtc`'s that have `self_refresh_active` in new state

Description

Called after `drm_mode_config_funcs.atomic_commit_tail`, this function will update the average entry/exit self refresh times on self refresh transitions. These averages will be used when calculating how long to delay before entering self refresh mode after activity.

void **drm_self_refresh_helper_alter_state**(struct *drm_atomic_state* *state)

Alters the atomic state for SR exit

Parameters

struct **drm_atomic_state** *state

the state currently being checked

Description

Called at the end of atomic check. This function checks the state for flags incompatible with self refresh exit and changes them. This is a bit disingenuous since userspace is expecting one thing and we're giving it another. However in order to keep self refresh entirely hidden from userspace, this is required.

At the end, we queue up the self refresh entry work so we can enter PSR after the desired delay.

int **drm_self_refresh_helper_init**(struct *drm_crtc* *crtc)

Initializes self refresh helpers for a crtc

Parameters

struct **drm_crtc** *crtc

the crtc which supports self refresh supported displays

Description

Returns zero if successful or -errno on failure

void **drm_self_refresh_helper_cleanup**(struct *drm_crtc* *crtc)

Cleans up self refresh helpers for a crtc

Parameters

struct **drm_crtc** *crtc

the crtc to cleanup

5.11 HDCP Helper Functions Reference

int **drm_hdcp_check_ksvs_revoked**(struct *drm_device* *drm_dev, u8 *ksvs, u32 ksv_count)

Check the revoked status of the IDs

Parameters

struct **drm_device** *drm_dev

drm_device for which HDCP revocation check is requested

u8 *ksvs

List of KSVs (HDCP receiver IDs)

u32 **ksv_count**

KSV count passed in through **ksvs**

Description

This function reads the HDCP System renewability Message(SRM Table) from userspace as a firmware and parses it for the revoked HDCP KSVs(Receiver IDs) detected by DCP LLC. Once the revoked KSVs are known, revoked state of the KSVs in the list passed in by display drivers are decided and response is sent.

SRM should be presented in the name of “display_hdcp_srm.bin”.

Format of the SRM table, that userspace needs to write into the binary file, is defined at: 1. Renewability chapter on 55th page of HDCP 1.4 specification [https://www.digital-cp.com/sites/default/files/specifications/HDCP`20Specification`20Rev1_4_Secure`.pdf](https://www.digital-cp.com/sites/default/files/specifications/HDCP%20Specification%20Rev1_4_Secure.pdf) 2. Renewability chapter on 63rd page of HDCP 2.2 specification [https://www.digital-cp.com/sites/default/files/specifications/HDCP`20on`20HDMI`20Specification`20Rev2_2_Final1`.pdf](https://www.digital-cp.com/sites/default/files/specifications/HDCP%20on%20HDMI%20Specification%20Rev2_2_Final1.pdf)

Return

Count of the revoked KSVs or -ve error number in case of the failure.

```
int drm_connector_attach_content_protection_property(struct drm_connector
                                                    *connector, bool
                                                    hdcp_content_type)
```

attach content protection property

Parameters

```
struct drm_connector *connector
connector to attach CP property on.
```

```
bool hdcp_content_type
is HDCP Content Type property needed for connector
```

Description

This is used to add support for content protection on select connectors. Content Protection is intentionally vague to allow for different underlying technologies, however it is most implemented by HDCP.

When `hdcp_content_type` is true enum property called HDCP Content Type is created (if it is not already) and attached to the connector.

This property is used for sending the protected content's stream type from userspace to kernel on selected connectors. Protected content provider will decide their type of their content and declare the same to kernel.

Content type will be used during the HDCP 2.2 authentication. Content type will be set to *drm_connector_state.hdcp_content_type*.

The content protection will be set to *drm_connector_state.content_protection*

When kernel triggered content protection state change like DESIRED->ENABLED and ENABLED->DESIRED, will use *drm_hdcp_update_content_protection()* to update the content protection state of a connector.

Return

Zero on success, negative errno on failure.

```
void drm_hdcp_update_content_protection(struct drm_connector *connector, u64 val)
Updates the content protection state of a connector
```

Parameters

struct drm_connector *connector

drm_connector on which content protection state needs an update

u64 val

New state of the content protection property

Description

This function can be used by display drivers, to update the kernel triggered content protection state changes of a drm_connector such as DESIRED->ENABLED and ENABLED->DESIRED. No uevent for DESIRED->UNDESIRED or ENABLED->UNDESIRED, as userspace is triggering such state change and kernel performs it without fail. This function update the new state of the property into the connector's state and generate an uevent to notify the userspace.

5.12 Display Port Helper Functions Reference

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

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

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

struct dp_sdp_header

DP secondary data packet header

Definition:

```
struct dp_sdp_header {
    u8 HB0;
    u8 HB1;
    u8 HB2;
    u8 HB3;
};
```

Members

HB0

Secondary Data Packet ID

HB1

Secondary Data Packet Type

HB2

Secondary Data Packet Specific header, Byte 0

HB3

Secondary Data packet Specific header, Byte 1

struct **dp_sdp**

DP secondary data packet

Definition:

```
struct dp_sdp {
    struct dp_sdp_header sdp_header;
    u8 db[32];
};
```

Members

sdp_header

DP secondary data packet header

db

DP secondary data packet data blocks VSC SDP Payload for PSR db[0]: Stereo Interface db[1]: 0 - PSR State; 1 - Update RFB; 2 - CRC Valid db[2]: CRC value bits 7:0 of the R or Cr component db[3]: CRC value bits 15:8 of the R or Cr component db[4]: CRC value bits 7:0 of the G or Y component db[5]: CRC value bits 15:8 of the G or Y component db[6]: CRC value bits 7:0 of the B or Cb component db[7]: CRC value bits 15:8 of the B or Cb component db[8] - db[31]: Reserved VSC SDP Payload for Pixel Encoding/Colorimetry Format db[0] - db[15]: Reserved db[16]: Pixel Encoding and Colorimetry Formats db[17]: Dynamic Range and Component Bit Depth db[18]: Content Type db[19] - db[31]: Reserved

enum **dp_pixelformat**

drm DP Pixel encoding formats

Constants

DP_PIXELFORMAT_RGB

RGB pixel encoding format

DP_PIXELFORMAT_YUV444

YCbCr 4:4:4 pixel encoding format

DP_PIXELFORMAT_YUV422

YCbCr 4:2:2 pixel encoding format

DP_PIXELFORMAT_YUV420

YCbCr 4:2:0 pixel encoding format

DP_PIXELFORMAT_Y_ONLY

Y Only pixel encoding format

DP_PIXELFORMAT_RAW

RAW pixel encoding format

DP_PIXELFORMAT_RESERVED

Reserved pixel encoding format

Description

This enum is used to indicate DP VSC SDP Pixel encoding formats. It is based on DP 1.4 spec [Table 2-117: VSC SDP Payload for DB16 through DB18]

enum **dp_colorimetry**

drm DP Colorimetry formats

Constants

DP_COLORIMETRY_DEFAULT

sRGB (IEC 61966-2-1) or ITU-R BT.601 colorimetry format

DP_COLORIMETRY_RGB_WIDE_FIXED

RGB wide gamut fixed point colorimetry format

DP_COLORIMETRY_BT709_YCC

ITU-R BT.709 colorimetry format

DP_COLORIMETRY_RGB_WIDE_FLOAT

RGB wide gamut floating point (scRGB (IEC 61966-2-2)) colorimetry format

DP_COLORIMETRY_XVYCC_601

xvYCC601 colorimetry format

DP_COLORIMETRY_OPRGB

OpRGB colorimetry format

DP_COLORIMETRY_XVYCC_709

xvYCC709 colorimetry format

DP_COLORIMETRY_DCI_P3_RGB

DCI-P3 (SMPTE RP 431-2) colorimetry format

DP_COLORIMETRY_SYCC_601

sYCC601 colorimetry format

DP_COLORIMETRY_RGB_CUSTOM

RGB Custom Color Profile colorimetry format

DP_COLORIMETRY_OPYCC_601

opYCC601 colorimetry format

DP_COLORIMETRY_BT2020_RGB

ITU-R BT.2020 R' G' B' colorimetry format

DP_COLORIMETRY_BT2020_CYCC

ITU-R BT.2020 Y'c C'bc C'rc colorimetry format

DP_COLORIMETRY_BT2020_YCC

ITU-R BT.2020 Y' C'b C'r colorimetry format

Description

This enum is used to indicate DP VSC SDP Colorimetry formats. It is based on DP 1.4 spec [Table 2-117: VSC SDP Payload for DB16 through DB18] and a name of enum member follows enum `drm_colorimetry` definition.

enum **dp_dynamic_range**

drm DP Dynamic Range

Constants

DP_DYNAMIC_RANGE_VESA

VESA range

DP_DYNAMIC_RANGE_CTA

CTA range

Description

This enum is used to indicate DP VSC SDP Dynamic Range. It is based on DP 1.4 spec [Table 2-117: VSC SDP Payload for DB16 through DB18]

enum **dp_content_type**
drm DP Content Type

Constants

DP_CONTENT_TYPE_NOT_DEFINED
Not defined type

DP_CONTENT_TYPE_GRAPHICS
Graphics type

DP_CONTENT_TYPE_PHOTO
Photo type

DP_CONTENT_TYPE_VIDEO
Video type

DP_CONTENT_TYPE_GAME
Game type

Description

This enum is used to indicate DP VSC SDP Content Types. It is based on DP 1.4 spec [Table 2-117: VSC SDP Payload for DB16 through DB18] CTA-861-G defines content types and expected processing by a sink device

struct **drm_dp_vsc_sdp**
drm DP VSC SDP

Definition:

```
struct drm_dp_vsc_sdp {
    unsigned char sdp_type;
    unsigned char revision;
    unsigned char length;
    enum dp_pixelformat pixelformat;
    enum dp_colorimetry colorimetry;
    int bpc;
    enum dp_dynamic_range dynamic_range;
    enum dp_content_type content_type;
};
```

Members

sdp_type
secondary-data packet type

revision
revision number

length
number of valid data bytes

pixelformat

pixel encoding format

colorimetry

colorimetry format

bpc

bit per color

dynamic_range

dynamic range information

content_type

CTA-861-G defines content types and expected processing by a sink device

Description

This structure represents a DP VSC SDP of drm It is based on DP 1.4 spec [Table 2-116: VSC SDP Header Bytes] and [Table 2-117: VSC SDP Payload for DB16 through DB18]

```
bool drm_dp_dsc_sink_supports_format(const u8
                                     dsc_dpcd[DP_DSC_RECEIVER_CAP_SIZE], u8
                                     output_format)
```

check if sink supports DSC with given output format

Parameters

```
const u8 dsc_dpcd[DP_DSC_RECEIVER_CAP_SIZE]
```

DSC-capability DPCDs of the sink

```
u8 output_format
```

output_format which is to be checked

Description

Returns true if the sink supports DSC with the given output_format, false otherwise.

```
bool drm_edp_backlight_supported(const u8 edp_dpcd[EDP_DISPLAY_CTL_CAP_SIZE])
```

Check an eDP DPCD for VESA backlight support

Parameters

```
const u8 edp_dpcd[EDP_DISPLAY_CTL_CAP_SIZE]
```

The DPCD to check

Description

Note that currently this function will return false for panels which support various DPCD backlight features but which require the brightness be set through PWM, and don't support setting the brightness level via the DPCD.

Return

True if **edp_dpcd** indicates that VESA backlight controls are supported, false otherwise

```
struct drm_dp_aux_msg
```

DisplayPort AUX channel transaction

Definition:

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

Members

address

address of the (first) register to access

request

contains the type of transaction (see DP_AUX_* macros)

reply

upon completion, contains the reply type of the transaction

buffer

pointer to a transmission or reception buffer

size

size of **buffer**

struct **drm_dp_aux_cec**

DisplayPort CEC-Tunneling-over-AUX

Definition:

```
struct drm_dp_aux_cec {
    struct mutex lock;
    struct cec_adapter *adap;
    struct drm_connector *connector;
    struct delayed_work unregister_work;
};
```

Members

lock

mutex protecting this struct

adap

the CEC adapter for CEC-Tunneling-over-AUX support.

connector

the connector this CEC adapter is associated with

unregister_work

unregister the CEC adapter

struct **drm_dp_aux**

DisplayPort AUX channel

Definition:

```
struct drm_dp_aux {
    const char *name;
    struct i2c_adapter ddc;
    struct device *dev;
    struct drm_device *drm_dev;
    struct drm_crtc *crtc;
    struct mutex hw_mutex;
    struct work_struct crc_work;
    u8 crc_count;
    ssize_t (*transfer)(struct drm_dp_aux *aux, struct drm_dp_aux_msg *msg);
    int (*wait_hpd_asserted)(struct drm_dp_aux *aux, unsigned long wait_us);
    unsigned i2c_nack_count;
    unsigned i2c_defer_count;
    struct drm_dp_aux_cec cec;
    bool is_remote;
};
```

Members

name

user-visible name of this AUX channel and the I2C-over-AUX adapter.

It's also used to specify the name of the I2C adapter. If set to NULL, `dev_name()` of **dev** will be used.

ddc

I2C adapter that can be used for I2C-over-AUX communication

dev

pointer to struct device that is the parent for this AUX channel.

drm_dev

pointer to the [drm_device](#) that owns this AUX channel. Beware, this may be NULL before [drm_dp_aux_register\(\)](#) has been called.

It should be set to the [drm_device](#) that will be using this AUX channel as early as possible. For many graphics drivers this should happen before [drm_dp_aux_init\(\)](#), however it's perfectly fine to set this field later so long as it's assigned before calling [drm_dp_aux_register\(\)](#).

crtc

backpointer to the crtc that is currently using this AUX channel

hw_mutex

internal mutex used for locking transfers.

Note that if the underlying hardware is shared among multiple channels, the driver needs to do additional locking to prevent concurrent access.

crc_work

worker that captures CRCs for each frame

crc_count

counter of captured frame CRCs

transfer

transfers a message representing a single AUX transaction.

This is a hardware-specific implementation of how transactions are executed that the drivers must provide.

A pointer to a `drm_dp_aux_msg` structure describing the transaction is passed into this function. Upon success, the implementation should return the number of payload bytes that were transferred, or a negative error-code on failure.

Helpers will propagate these errors, with the exception of the `-EBUSY` error, which causes a transaction to be retried. On a short, helpers will return `-EPROTO` to make it simpler to check for failure.

The **transfer()** function must only modify the reply field of the `drm_dp_aux_msg` structure. The retry logic and i2c helpers assume this is the case.

Also note that this callback can be called no matter the state **dev** is in and also no matter what state the panel is in. It's expected:

- If the **dev** providing the AUX bus is currently unpowered then it will power itself up for the transfer.
- If we're on eDP (using a `drm_panel`) and the panel is not in a state where it can respond (it's not powered or it's in a low power state) then this function may return an error, but not crash. It's up to the caller of this code to make sure that the panel is powered on if getting an error back is not OK. If a `drm_panel` driver is initiating a DP AUX transfer it may power itself up however it wants. All other code should ensure that the `pre_enable()` bridge chain (which eventually calls the `drm_panel` prepare function) has powered the panel.

wait_hpd_asserted

wait for HPD to be asserted

This is mainly useful for eDP panels drivers to wait for an eDP panel to finish powering on. This is an optional function.

This function will efficiently wait for the HPD signal to be asserted. The `wait_us` parameter that is passed in says that we know that the HPD signal is expected to be asserted within `wait_us` microseconds. This function could wait for longer than `wait_us` if the logic in the DP controller has a long debouncing time. The important thing is that if this function returns success that the DP controller is ready to send AUX transactions.

This function returns 0 if HPD was asserted or `-ETIMEDOUT` if time expired and HPD wasn't asserted. This function should not print timeout errors to the log.

The semantics of this function are designed to match the `readx_poll_timeout()` function. That means a `wait_us` of 0 means to wait forever. Like `readx_poll_timeout()`, this function may sleep.

NOTE: this function specifically reports the state of the HPD pin that's associated with the DP AUX channel. This is different from the HPD concept in much of the rest of DRM which is more about physical presence of a display. For eDP, for instance, a display is assumed always present even if the HPD pin is deasserted.

i2c_nack_count

Counts I2C NACKs, used for DP validation.

i2c_defer_count

Counts I2C DEFERs, used for DP validation.

cec

struct containing fields used for CEC-Tunneling-over-AUX.

is_remote

Is this AUX CH actually using sideband messaging.

Description

An AUX channel can also be used to transport I2C messages to a sink. A typical application of that is to access an EDID that's present in the sink device. The **transfer()** function can also be used to execute such transactions. The `drm_dp_aux_register()` function registers an I2C adapter that can be passed to `drm_probe_ddc()`. Upon removal, drivers should call `drm_dp_aux_unregister()` to remove the I2C adapter. The I2C adapter uses long transfers by default; if a partial response is received, the adapter will drop down to the size given by the partial response for this transaction only.

`ssize_t drm_dp_dpcd_readb(struct drm_dp_aux *aux, unsigned int offset, u8 *valuep)`
read a single byte from the DPCD

Parameters

struct *drm_dp_aux* *aux
DisplayPort AUX channel

unsigned int offset
address of the register to read

u8 *valuep
location where the value of the register will be stored

Description

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

`ssize_t drm_dp_dpcd_writeb(struct drm_dp_aux *aux, unsigned int offset, u8 value)`
write a single byte to the DPCD

Parameters

struct *drm_dp_aux* *aux
DisplayPort AUX channel

unsigned int offset
address of the register to write

u8 value
value to write to the register

Description

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

struct *drm_dp_desc*
DP branch/sink device descriptor

Definition:

```
struct drm_dp_desc {  
    struct drm_dp_dpcd_ident ident;
```

```
    u32 quirks;
};
```

Members

ident

DP device identification from DPCD 0x400 (sink) or 0x500 (branch).

quirks

Quirks; use [drm_dp_has_quirk\(\)](#) to query for the quirks.

enum **drm_dp_quirk**

Display Port sink/branch device specific quirks

Constants

DP_DPCD_QUIRK_CONSTANT_N

The device requires main link attributes Mvid and Nvid to be limited to 16 bits. So will give a constant value (0x8000) for compatability.

DP_DPCD_QUIRK_NO_PSR

The device does not support PSR even if reports that it supports or driver still need to implement proper handling for such device.

DP_DPCD_QUIRK_NO_SINK_COUNT

The device does not set SINK_COUNT to a non-zero value. The driver should ignore SINK_COUNT during detection. Note that [drm_dp_read_sink_count_cap\(\)](#) automatically checks for this quirk.

DP_DPCD_QUIRK_DSC_WITHOUT_VIRTUAL_DPCD

The device supports MST DSC despite not supporting Virtual DPCD. The DSC caps can be read from the physical aux instead.

DP_DPCD_QUIRK_CAN_DO_MAX_LINK_RATE_3_24_GBPS

The device supports a link rate of 3.24 Gbps (multiplier 0xc) despite the DP_MAX_LINK_RATE register reporting a lower max multiplier.

Description

Display Port sink and branch devices in the wild have a variety of bugs, try to collect them here. The quirks are shared, but it's up to the drivers to implement workarounds for them.

bool **drm_dp_has_quirk**(const struct [drm_dp_desc](#) *desc, enum [drm_dp_quirk](#) quirk)
does the DP device have a specific quirk

Parameters

const struct **drm_dp_desc** *desc

Device descriptor filled by [drm_dp_read_desc\(\)](#)

enum **drm_dp_quirk** quirk

Quirk to query for

Description

Return true if DP device identified by **desc** has **quirk**.

struct **drm_edp_backlight_info**

Probed eDP backlight info struct

Definition:

```
struct drm_edp_backlight_info {
    u8 pwmgen_bit_count;
    u8 pwm_freq_pre_divider;
    u16 max;
    bool lsb_reg_used : 1;
    bool aux_enable : 1;
    bool aux_set : 1;
};
```

Members

pwmgen_bit_count

The pwmgen bit count

pwm_freq_pre_divider

The PWM frequency pre-divider value being used for this backlight, if any

max

The maximum backlight level that may be set

lsb_reg_used

Do we also write values to the DP_EDP_BACKLIGHT_BRIGHTNESS_LSB register?

aux_enable

Does the panel support the AUX enable cap?

aux_set

Does the panel support setting the brightness through AUX?

Description

This structure contains various data about an eDP backlight, which can be populated by using [*drm_edp_backlight_init\(\)*](#).

struct drm_dp_phy_test_params

DP Phy Compliance parameters

Definition:

```
struct drm_dp_phy_test_params {
    int link_rate;
    u8 num_lanes;
    u8 phy_pattern;
    u8 hbr2_reset[2];
    u8 custom80[10];
    bool enhanced_frame_cap;
};
```

Members

link_rate

Requested Link rate from DPCD 0x219

num_lanes

Number of lanes requested by sing through DPCD 0x220

phy_pattern

DP Phy test pattern from DPCD 0x248

hbr2_reset

DP HBR2_COMPLIANCE_SCRAMBLER_RESET from DCPD 0x24A and 0x24B

custom80

DP Test_80BIT_CUSTOM_PATTERN from DPCDs 0x250 through 0x259

enhanced_frame_cap

flag for enhanced frame capability.

const char ***drm_dp_phy_name**(enum drm_dp_phy dp_phy)

Get the name of the given DP PHY

Parameters

enum **drm_dp_phy dp_phy**

The DP PHY identifier

Description

Given the **dp_phy**, get a user friendly name of the DP PHY, either "DPRX" or "LTTPR <N>", or "<INVALID DP PHY>" on errors. The returned string is always non-NULL and valid.

Return

Name of the DP PHY.

int **drm_dp_dpcd_probe**(struct *drm_dp_aux* *aux, unsigned int offset)

probe a given DPCD address with a 1-byte read access

Parameters

struct **drm_dp_aux *aux**

DisplayPort AUX channel (SST)

unsigned int **offset**

address of the register to probe

Description

Probe the provided DPCD address by reading 1 byte from it. The function can be used to trigger some side-effect the read access has, like waking up the sink, without the need for the read-out value.

Returns 0 if the read access succeeded, or a negative error code on failure.

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

read a series of bytes from the DPCD

Parameters

struct **drm_dp_aux *aux**

DisplayPort AUX channel (SST or MST)

unsigned int **offset**

address of the (first) register to read

void ***buffer**

buffer to store the register values

size_t size

number of bytes in **buffer**

Description

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

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

write a series of bytes to the DPCD

Parameters

struct drm_dp_aux *aux

DisplayPort AUX channel (SST or MST)

unsigned int offset

address of the (first) register to write

void *buffer

buffer containing the values to write

size_t size

number of bytes in **buffer**

Description

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

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

read DPCD link status (bytes 0x202-0x207)

Parameters

struct drm_dp_aux *aux

DisplayPort AUX channel

u8 status[DP_LINK_STATUS_SIZE]

buffer to store the link status in (must be at least 6 bytes)

Description

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

int **drm_dp_dpcd_read_phy_link_status**(struct *drm_dp_aux* *aux, enum drm_dp_phy dp_phy, u8 link_status[DP_LINK_STATUS_SIZE])

get the link status information for a DP PHY

Parameters

```

struct drm_dp_aux *aux
    DisplayPort AUX channel

enum drm_dp_phy dp_phy
    the DP PHY to get the link status for

u8 link_status[DP_LINK_STATUS_SIZE]
    buffer to return the status in

```

Description

Fetch the AUX DPCD registers for the DPRX or an LTTPR PHY link status. The layout of the returned **link_status** matches the DPCD register layout of the DPRX PHY link status.

Returns 0 if the information was read successfully or a negative error code on failure.

```

bool drm_dp_downstream_is_type(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8
                                port_cap[4], u8 type)

    is the downstream facing port of certain type?

```

Parameters

```

const u8 dpcd[DP_RECEIVER_CAP_SIZE]
    DisplayPort configuration data

const u8 port_cap[4]
    port capabilities

u8 type
    port type to be checked. Can be: DP_DS_PORT_TYPE_DP, DP_DS_PORT_TYPE_VGA,
    DP_DS_PORT_TYPE_DVI, DP_DS_PORT_TYPE_HDMI, DP_DS_PORT_TYPE_NON_EDID,
    DP_DS_PORT_TYPE_DP_DUALMODE or DP_DS_PORT_TYPE_WIRELESS.

```

Description

Caveat: Only works with DPCD 1.1+ port caps.

Return

whether the downstream facing port matches the type.

```

bool drm_dp_downstream_is_tmds(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8
                                port_cap[4], const struct edid *edid)

    is the downstream facing port TMDS?

```

Parameters

```

const u8 dpcd[DP_RECEIVER_CAP_SIZE]
    DisplayPort configuration data

const u8 port_cap[4]
    port capabilities

const struct edid *edid
    EDID

```

Return

whether the downstream facing port is TMDS (HDMI/DVI).

bool **drm_dp_send_real_edid_checksum**(struct *drm_dp_aux* *aux, u8 real_edid_checksum)
send back real edid checksum value

Parameters

struct drm_dp_aux *aux
DisplayPort AUX channel

u8 real_edid_checksum
real edid checksum for the last block

Return

True on success

int **drm_dp_read_dpcd_caps**(struct *drm_dp_aux* *aux, u8 dpcd[DP_RECEIVER_CAP_SIZE])
read DPCD caps and extended DPCD caps if available

Parameters

struct drm_dp_aux *aux
DisplayPort AUX channel

u8 dpcd[DP_RECEIVER_CAP_SIZE]
Buffer to store the resulting DPCD in

Description

Attempts to read the base DPCD caps for **aux**. Additionally, this function checks for and reads the extended DPRX caps (DP_DP13_DPCD_REV) if present.

Return

0 if the DPCD was read successfully, negative error code otherwise.

int **drm_dp_read_downstream_info**(struct *drm_dp_aux* *aux, const u8
dpcd[DP_RECEIVER_CAP_SIZE], u8
downstream_ports[DP_MAX_DOWNSTREAM_PORTS])
read DPCD downstream port info if available

Parameters

struct drm_dp_aux *aux
DisplayPort AUX channel

const u8 dpcd[DP_RECEIVER_CAP_SIZE]
A cached copy of the port's DPCD

u8 downstream_ports[DP_MAX_DOWNSTREAM_PORTS]
buffer to store the downstream port info in

Description

See also: `drm_dp_downstream_max_clock()` *drm_dp_downstream_max_bpc()*

Return

0 if either the downstream port info was read successfully or there was no downstream info to read, or a negative error code otherwise.


```
int drm_dp_downstream_max_dotclock(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8
                                   port_cap[4])
```

extract downstream facing port max dot clock

Parameters

```
const u8 dpcd[DP_RECEIVER_CAP_SIZE]
```

DisplayPort configuration data

```
const u8 port_cap[4]
```

port capabilities

Return

Downstream facing port max dot clock in kHz on success, or 0 if max clock not defined

```
int drm_dp_downstream_max_tmds_clock(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8
                                   port_cap[4], const struct edid *edid)
```

extract downstream facing port max TMDS clock

Parameters

```
const u8 dpcd[DP_RECEIVER_CAP_SIZE]
```

DisplayPort configuration data

```
const u8 port_cap[4]
```

port capabilities

```
const struct edid *edid
```

EDID

Return

HDMI/DVI downstream facing port max TMDS clock in kHz on success, or 0 if max TMDS clock not defined

```
int drm_dp_downstream_min_tmds_clock(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8
                                   port_cap[4], const struct edid *edid)
```

extract downstream facing port min TMDS clock

Parameters

```
const u8 dpcd[DP_RECEIVER_CAP_SIZE]
```

DisplayPort configuration data

```
const u8 port_cap[4]
```

port capabilities

```
const struct edid *edid
```

EDID

Return

HDMI/DVI downstream facing port min TMDS clock in kHz on success, or 0 if max TMDS clock not defined

```
int drm_dp_downstream_max_bpc(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8
                               port_cap[4], const struct edid *edid)
```

extract downstream facing port max bits per component

Parameters

const u8 dpcd[DP_RECEIVER_CAP_SIZE]

DisplayPort configuration data

const u8 port_cap[4]

downstream facing port capabilities

const struct edid *edid

EDID

Return

Max bpc on success or 0 if max bpc not defined

bool **drm_dp_downstream_420_passthrough**(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8 port_cap[4])

determine downstream facing port YCbCr 4:2:0 pass-through capability

Parameters

const u8 dpcd[DP_RECEIVER_CAP_SIZE]

DisplayPort configuration data

const u8 port_cap[4]

downstream facing port capabilities

Return

whether the downstream facing port can pass through YCbCr 4:2:0

bool **drm_dp_downstream_444_to_420_conversion**(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8 port_cap[4])

determine downstream facing port YCbCr 4:4:4->4:2:0 conversion capability

Parameters

const u8 dpcd[DP_RECEIVER_CAP_SIZE]

DisplayPort configuration data

const u8 port_cap[4]

downstream facing port capabilities

Return

whether the downstream facing port can convert YCbCr 4:4:4 to 4:2:0

bool **drm_dp_downstream_rgb_to_ycbcr_conversion**(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8 port_cap[4], u8 color_spc)

determine downstream facing port RGB->YCbCr conversion capability

Parameters

const u8 dpcd[DP_RECEIVER_CAP_SIZE]

DisplayPort configuration data

const u8 port_cap[4]

downstream facing port capabilities

u8 color_spc

Colorspace for which conversion cap is sought

Return

whether the downstream facing port can convert RGB->YCbCr for a given colorspace.

```
struct drm_display_mode *drm_dp_downstream_mode(struct drm_device *dev, const u8
                                              dpcd[DP_RECEIVER_CAP_SIZE], const
                                              u8 port_cap[4])
```

return a mode for downstream facing port

Parameters

```
struct drm_device *dev
    DRM device
```

```
const u8 dpcd[DP_RECEIVER_CAP_SIZE]
    DisplayPort configuration data
```

```
const u8 port_cap[4]
    port capabilities
```

Description

Provides a suitable mode for downstream facing ports without EDID.

Return

A new *drm_display_mode* on success or NULL on failure

```
int drm_dp_downstream_id(struct drm_dp_aux *aux, char id[6])
    identify branch device
```

Parameters

```
struct drm_dp_aux *aux
    DisplayPort AUX channel
```

```
char id[6]
    DisplayPort branch device id
```

Description

Returns branch device id on success or NULL on failure

```
void drm_dp_downstream_debug(struct seq_file *m, const u8 dpcd[DP_RECEIVER_CAP_SIZE],
                             const u8 port_cap[4], const struct edid *edid, struct
                             drm_dp_aux *aux)
```

debug DP branch devices

Parameters

```
struct seq_file *m
    pointer for debugfs file
```

```
const u8 dpcd[DP_RECEIVER_CAP_SIZE]
    DisplayPort configuration data
```

```
const u8 port_cap[4]
    port capabilities
```

```
const struct edid *edid
    EDID
```

struct drm_dp_aux *aux

DisplayPort AUX channel

enum drm_mode_subconnector **drm_dp_subconnector_type**(const u8
dpcd[DP_RECEIVER_CAP_SIZE],
const u8 port_cap[4])

get DP branch device type

Parameters

const u8 dpcd[DP_RECEIVER_CAP_SIZE]

DisplayPort configuration data

const u8 port_cap[4]

port capabilities

void **drm_dp_set_subconnector_property**(struct *drm_connector* *connector, enum
drm_connector_status status, const u8 *dpcd,
const u8 port_cap[4])

set subconnector for DP connector

Parameters

struct drm_connector *connector

connector to set property on

enum drm_connector_status status

connector status

const u8 *dpcd

DisplayPort configuration data

const u8 port_cap[4]

port capabilities

Description

Called by a driver on every detect event.

bool **drm_dp_read_sink_count_cap**(struct *drm_connector* *connector, const u8
dpcd[DP_RECEIVER_CAP_SIZE], const struct
drm_dp_desc *desc)

Check whether a given connector has a valid sink count

Parameters

struct drm_connector *connector

The DRM connector to check

const u8 dpcd[DP_RECEIVER_CAP_SIZE]

A cached copy of the connector's DPCD RX capabilities

const struct drm_dp_desc *desc

A cached copy of the connector's DP descriptor

Description

See also: *drm_dp_read_sink_count()*

Return

True if the (e)DP connector has a valid sink count that should be probed, false otherwise.

int **drm_dp_read_sink_count**(struct *drm_dp_aux* *aux)

Retrieve the sink count for a given sink

Parameters

struct drm_dp_aux *aux

The DP AUX channel to use

Description

See also: *drm_dp_read_sink_count_cap()*

Return

The current sink count reported by **aux**, or a negative error code otherwise.

void **drm_dp_remote_aux_init**(struct *drm_dp_aux* *aux)

minimally initialise a remote aux channel

Parameters

struct drm_dp_aux *aux

DisplayPort AUX channel

Description

Used for remote aux channel in general. Merely initialize the crc work struct.

void **drm_dp_aux_init**(struct *drm_dp_aux* *aux)

minimally initialise an aux channel

Parameters

struct drm_dp_aux *aux

DisplayPort AUX channel

Description

If you need to use the *drm_dp_aux*'s i2c adapter prior to registering it with the outside world, call *drm_dp_aux_init()* first. For drivers which are grandparents to their AUX adapters (e.g. the AUX adapter is parented by a *drm_connector*), you must still call *drm_dp_aux_register()* once the connector has been registered to allow userspace access to the auxiliary DP channel. Likewise, for such drivers you should also assign *drm_dp_aux.drm_dev* as early as possible so that the *drm_device* that corresponds to the AUX adapter may be mentioned in debugging output from the DRM DP helpers.

For devices which use a separate platform device for their AUX adapters, this may be called as early as required by the driver.

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

initialise and register aux channel

Parameters

struct drm_dp_aux *aux

DisplayPort AUX channel

Description

Automatically calls `drm_dp_aux_init()` if this hasn't been done yet. This should only be called once the parent of **aux**, `drm_dp_aux.dev`, is initialized. For devices which are grandparents of their AUX channels, `drm_dp_aux.dev` will typically be the `drm_connector` device which corresponds to **aux**. For these devices, it's advised to call `drm_dp_aux_register()` in `drm_connector_funcs.late_register`, and likewise to call `drm_dp_aux_unregister()` in `drm_connector_funcs.early_unregister`. Functions which don't follow this will likely Oops when `CONFIG_DRM_DP_AUX_CHARDEV` is enabled.

For devices where the AUX channel is a device that exists independently of the `drm_device` that uses it, such as SoCs and bridge devices, it is recommended to call `drm_dp_aux_register()` after a `drm_device` has been assigned to `drm_dp_aux.drm_dev`, and likewise to call `drm_dp_aux_unregister()` once the `drm_device` should no longer be associated with the AUX channel (e.g. on bridge detach).

Drivers which need to use the aux channel before either of the two points mentioned above need to call `drm_dp_aux_init()` in order to use the AUX channel before registration.

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

void **drm_dp_aux_unregister**(struct `drm_dp_aux` *aux)
unregister an AUX adapter

Parameters

struct **drm_dp_aux** *aux
DisplayPort AUX channel

int **drm_dp_psr_setup_time**(const u8 psr_cap[EDP_PSR_RECEIVER_CAP_SIZE])
PSR setup in time usec

Parameters

const u8 **psr_cap**[EDP_PSR_RECEIVER_CAP_SIZE]
PSR capabilities from DPCD

Return

PSR setup time for the panel in microseconds, negative error code on failure.

int **drm_dp_start_crc**(struct `drm_dp_aux` *aux, struct `drm_crtc` *crtc)
start capture of frame CRCs

Parameters

struct **drm_dp_aux** *aux
DisplayPort AUX channel

struct **drm_crtc** *crtc
CRTC displaying the frames whose CRCs are to be captured

Description

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

int **drm_dp_stop_crc**(struct `drm_dp_aux` *aux)
stop capture of frame CRCs

Parameters

struct **drm_dp_aux** *aux
DisplayPort AUX channel

Description

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

int **drm_dp_read_desc**(struct *drm_dp_aux* *aux, struct *drm_dp_desc* *desc, bool is_branch)
 read sink/branch descriptor from DPCD

Parameters

struct drm_dp_aux *aux
 DisplayPort AUX channel

struct drm_dp_desc *desc
 Device descriptor to fill from DPCD

bool is_branch
 true for branch devices, false for sink devices

Description

Read DPCD 0x400 (sink) or 0x500 (branch) into **desc**. Also debug log the identification.

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

u8 **drm_dp_dsc_sink_max_slice_count**(const u8 dsc_dpcd[DP_DSC_RECEIVER_CAP_SIZE],
 bool is_edp)
 Get the max slice count supported by the DSC sink.

Parameters

const u8 dsc_dpcd[DP_DSC_RECEIVER_CAP_SIZE]
 DSC capabilities from DPCD

bool is_edp
 true if its eDP, false for DP

Description

Read the slice capabilities DPCD register from DSC sink to get the maximum slice count supported. This is used to populate the DSC parameters in the *struct drm_dsc_config* by the driver. Driver creates an infoframe using these parameters to populate *struct drm_dsc_pps_infoframe*. These are sent to the sink using DSC infoframe using the helper function `drm_dsc_pps_infoframe_pack()`

Return

Maximum slice count supported by DSC sink or 0 its invalid

u8 **drm_dp_dsc_sink_line_buf_depth**(const u8 dsc_dpcd[DP_DSC_RECEIVER_CAP_SIZE])
 Get the line buffer depth in bits

Parameters

const u8 dsc_dpcd[DP_DSC_RECEIVER_CAP_SIZE]
 DSC capabilities from DPCD

Description

Read the DSC DPCD register to parse the line buffer depth in bits which is number of bits of precision within the decoder line buffer supported by the DSC sink. This is used to populate the DSC parameters in the *struct drm_dsc_config* by the driver. Driver creates an infoframe

using these parameters to populate *struct drm_dsc_pps_infotrame*. These are sent to the sink using DSC infotrame using the helper function `drm_dsc_pps_infotrame_pack()`

Return

Line buffer depth supported by DSC panel or 0 its invalid

```
int drm_dp_dsc_sink_supported_input_bpcs(const u8
                                         dsc_dpcd[DP_DSC_RECEIVER_CAP_SIZE], u8
                                         dsc_bpc[3])
```

Get all the input bits per component values supported by the DSC sink.

Parameters

```
const u8 dsc_dpcd[DP_DSC_RECEIVER_CAP_SIZE]
    DSC capabilities from DPCD
```

```
u8 dsc_bpc[3]
    An array to be filled by this helper with supported input bpcs.
```

Description

Read the DSC DPCD from the sink device to parse the supported bits per component values. This is used to populate the DSC parameters in the *struct drm_dsc_config* by the driver. Driver creates an infotrame using these parameters to populate *struct drm_dsc_pps_infotrame*. These are sent to the sink using DSC infotrame using the helper function `drm_dsc_pps_infotrame_pack()`

Return

Number of input BPC values parsed from the DPCD

```
int drm_dp_read_lttpr_common_caps(struct drm_dp_aux *aux, const u8
                                   dpcd[DP_RECEIVER_CAP_SIZE], u8
                                   caps[DP_LTTPR_COMMON_CAP_SIZE])

    read the LTTPR common capabilities
```

Parameters

```
struct drm_dp_aux *aux
    DisplayPort AUX channel

const u8 dpcd[DP_RECEIVER_CAP_SIZE]
    DisplayPort configuration data

u8 caps[DP_LTTPR_COMMON_CAP_SIZE]
    buffer to return the capability info in
```

Description

Read capabilities common to all LTTPRs.

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

```
int drm_dp_read_lttpr_phy_caps(struct drm_dp_aux *aux, const u8
                                dpcd[DP_RECEIVER_CAP_SIZE], enum drm_dp_phy
                                dp_phy, u8 caps[DP_LTTPR_PHY_CAP_SIZE])

    read the capabilities for a given LTTPR PHY
```

Parameters


```
struct drm_dp_aux *aux
    DisplayPort AUX channel

const u8 dpcd[DP_RECEIVER_CAP_SIZE]
    DisplayPort configuration data

enum drm_dp_phy dp_phy
    LTTPR PHY to read the capabilities for

u8 caps[DP_LTTPR_PHY_CAP_SIZE]
    buffer to return the capability info in
```

Description

Read the capabilities for the given LTTPR PHY.

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

```
int drm_dp_lttpr_count(const u8 caps[DP_LTTPR_COMMON_CAP_SIZE])
    get the number of detected LTTPRs
```

Parameters

```
const u8 caps[DP_LTTPR_COMMON_CAP_SIZE]
    LTTPR common capabilities
```

Description

Get the number of detected LTTPRs from the LTTPR common capabilities info.

Return

-ERANGE if more than supported number (8) of LTTPRs are detected -EINVAL if the DP_PHY_REPEATER_CNT register contains an invalid value otherwise the number of detected LTTPRs

```
int drm_dp_lttpr_max_link_rate(const u8 caps[DP_LTTPR_COMMON_CAP_SIZE])
    get the maximum link rate supported by all LTTPRs
```

Parameters

```
const u8 caps[DP_LTTPR_COMMON_CAP_SIZE]
    LTTPR common capabilities
```

Description

Returns the maximum link rate supported by all detected LTTPRs.

```
int drm_dp_lttpr_max_lane_count(const u8 caps[DP_LTTPR_COMMON_CAP_SIZE])
    get the maximum lane count supported by all LTTPRs
```

Parameters

```
const u8 caps[DP_LTTPR_COMMON_CAP_SIZE]
    LTTPR common capabilities
```

Description

Returns the maximum lane count supported by all detected LTTPRs.

bool **drm_dp_lttpr_voltage_swing_level_3_supported**(const u8
caps[DP_LTTPR_PHY_CAP_SIZE])
check for LTTPR vswing3 support

Parameters

const u8 caps[DP_LTTPR_PHY_CAP_SIZE]
LTTPR PHY capabilities

Description

Returns true if the **caps** for an LTTPR TX PHY indicate support for voltage swing level 3.

bool **drm_dp_lttpr_pre_emphasis_level_3_supported**(const u8
caps[DP_LTTPR_PHY_CAP_SIZE])
check for LTTPR preemph3 support

Parameters

const u8 caps[DP_LTTPR_PHY_CAP_SIZE]
LTTPR PHY capabilities

Description

Returns true if the **caps** for an LTTPR TX PHY indicate support for pre-emphasis level 3.

int **drm_dp_get_phy_test_pattern**(struct *drm_dp_aux* *aux, struct *drm_dp_phy_test_params*
*data)
get the requested pattern from the sink.

Parameters

struct *drm_dp_aux* *aux
DisplayPort AUX channel

struct *drm_dp_phy_test_params* *data
DP phy compliance test parameters.

Description

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

int **drm_dp_set_phy_test_pattern**(struct *drm_dp_aux* *aux, struct *drm_dp_phy_test_params*
*data, u8 dp_rev)
set the pattern to the sink.

Parameters

struct *drm_dp_aux* *aux
DisplayPort AUX channel

struct *drm_dp_phy_test_params* *data
DP phy compliance test parameters.

u8 dp_rev
DP revision to use for compliance testing

Description

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

int **drm_dp_get_pcon_max_frl_bw**(const u8 dpcd[DP_RECEIVER_CAP_SIZE], const u8 port_cap[4])

maximum frl supported by PCON

Parameters

const u8 dpcd[DP_RECEIVER_CAP_SIZE]

DisplayPort configuration data

const u8 port_cap[4]

port capabilities

Description

Returns maximum frl bandwidth supported by PCON in GBPS, returns 0 if not supported.

int **drm_dp_pcon_frl_prepare**(struct *drm_dp_aux* *aux, bool enable_frl_ready_hpd)

Prepare PCON for FRL.

Parameters

struct *drm_dp_aux* *aux

DisplayPort AUX channel

bool enable_frl_ready_hpd

Configure DP_PCON_ENABLE_HPD_READY.

Description

Returns 0 if success, else returns negative error code.

bool **drm_dp_pcon_is_frl_ready**(struct *drm_dp_aux* *aux)

Is PCON ready for FRL

Parameters

struct *drm_dp_aux* *aux

DisplayPort AUX channel

Description

Returns true if success, else returns false.

int **drm_dp_pcon_frl_configure_1**(struct *drm_dp_aux* *aux, int max_frl_gbps, u8 frl_mode)

Set HDMI LINK Configuration-Step1

Parameters

struct *drm_dp_aux* *aux

DisplayPort AUX channel

int max_frl_gbps

maximum frl bw to be configured between PCON and HDMI sink

u8 frl_mode

FRL Training mode, it can be either Concurrent or Sequential. In Concurrent Mode, the FRL link bring up can be done along with DP Link training. In Sequential mode, the FRL link bring up is done prior to the DP Link training.

Description

Returns 0 if success, else returns negative error code.

int **drm_dp_pcon_frl_configure_2**(struct *drm_dp_aux* *aux, int max_frl_mask, u8 frl_type)

Set HDMI Link configuration Step-2

Parameters

struct *drm_dp_aux* *aux

DisplayPort AUX channel

int max_frl_mask

Max FRL BW to be tried by the PCON with HDMI Sink

u8 frl_type

FRL training type, can be Extended, or Normal. In Normal FRL training, the PCON tries each frl bw from the max_frl_mask starting from min, and stops when link training is successful. In Extended FRL training, all frl bw selected in the mask are trained by the PCON.

Description

Returns 0 if success, else returns negative error code.

int **drm_dp_pcon_reset_frl_config**(struct *drm_dp_aux* *aux)

Re-Set HDMI Link configuration.

Parameters

struct *drm_dp_aux* *aux

DisplayPort AUX channel

Description

Returns 0 if success, else returns negative error code.

int **drm_dp_pcon_frl_enable**(struct *drm_dp_aux* *aux)

Enable HDMI link through FRL

Parameters

struct *drm_dp_aux* *aux

DisplayPort AUX channel

Description

Returns 0 if success, else returns negative error code.

bool **drm_dp_pcon_hdmi_link_active**(struct *drm_dp_aux* *aux)

check if the PCON HDMI LINK status is active.

Parameters

struct *drm_dp_aux* *aux

DisplayPort AUX channel

Description

Returns true if link is active else returns false.

int **drm_dp_pcon_hdmi_link_mode**(struct *drm_dp_aux* *aux, u8 *frl_trained_mask)

get the PCON HDMI LINK MODE

Parameters

struct drm_dp_aux *aux

DisplayPort AUX channel

u8 *frl_trained_mask

pointer to store bitmask of the trained bw configuration. Valid only if the MODE returned is FRL. For Normal Link training mode only 1 of the bits will be set, but in case of Extended mode, more than one bits can be set.

Description

Returns the link mode : TMDS or FRL on success, else returns negative error code.

void **drm_dp_pcon_hdmi_frl_link_error_count**(struct *drm_dp_aux* *aux, struct *drm_connector* *connector)

print the error count per lane during link failure between PCON and HDMI sink

Parameters

struct drm_dp_aux *aux

DisplayPort AUX channel

struct drm_connector *connector

DRM connector code.

int **drm_dp_pcon_pps_default**(struct *drm_dp_aux* *aux)

Let PCON fill the default pps parameters for DSC1.2 between PCON & HDMI2.1 sink

Parameters

struct drm_dp_aux *aux

DisplayPort AUX channel

Description

Returns 0 on success, else returns negative error code.

int **drm_dp_pcon_pps_override_buf**(struct *drm_dp_aux* *aux, u8 pps_buf[128])

Configure PPS encoder override buffer for HDMI sink

Parameters

struct drm_dp_aux *aux

DisplayPort AUX channel

u8 pps_buf[128]

128 bytes to be written into PPS buffer for HDMI sink by PCON.

Description

Returns 0 on success, else returns negative error code.

int **drm_edp_backlight_set_level**(struct *drm_dp_aux* *aux, const struct *drm_edp_backlight_info* *bl, u16 level)

Set the backlight level of an eDP panel via AUX

Parameters

struct drm_dp_aux *aux

The DP AUX channel to use

const struct drm_edp_backlight_info *bl

Backlight capability info from `drm_edp_backlight_init()`

u16 level

The brightness level to set

Description

Sets the brightness level of an eDP panel's backlight. Note that the panel's backlight must already have been enabled by the driver by calling `drm_edp_backlight_enable()`.

Return

0 on success, negative error code on failure

int **drm_edp_backlight_enable**(struct `drm_dp_aux` *aux, const struct `drm_edp_backlight_info` *bl, const u16 level)

Enable an eDP panel's backlight using DPCD

Parameters

struct drm_dp_aux *aux

The DP AUX channel to use

const struct drm_edp_backlight_info *bl

Backlight capability info from `drm_edp_backlight_init()`

const u16 level

The initial backlight level to set via AUX, if there is one

Description

This function handles enabling DPCD backlight controls on a panel over DPCD, while additionally restoring any important backlight state such as the given backlight level, the brightness byte count, backlight frequency, etc.

Note that certain panels do not support being enabled or disabled via DPCD, but instead require that the driver handle enabling/disabling the panel through implementation-specific means using the EDP_BL_PWR GPIO. For such panels, `drm_edp_backlight_info.aux_enable` will be set to false, this function becomes a no-op, and the driver is expected to handle powering the panel on using the EDP_BL_PWR GPIO.

Return

0 on success, negative error code on failure.

int **drm_edp_backlight_disable**(struct `drm_dp_aux` *aux, const struct `drm_edp_backlight_info` *bl)

Disable an eDP backlight using DPCD, if supported

Parameters

struct drm_dp_aux *aux

The DP AUX channel to use

const struct drm_edp_backlight_info *bl

Backlight capability info from `drm_edp_backlight_init()`

Description

This function handles disabling DPCD backlight controls on a panel over AUX.

Note that certain panels do not support being enabled or disabled via DPCD, but instead require that the driver handle enabling/disabling the panel through implementation-specific means using the EDP_BL_PWR GPIO. For such panels, *drm_edp_backlight_info.aux_enable* will be set to false, this function becomes a no-op, and the driver is expected to handle powering the panel off using the EDP_BL_PWR GPIO.

Return

0 on success or no-op, negative error code on failure.

```
int drm_edp_backlight_init(struct drm_dp_aux *aux, struct drm_edp_backlight_info *bl, u16
                           driver_pwm_freq_hz, const u8
                           edp_dpcd[EDP_DISPLAY_CTL_CAP_SIZE], u16 *current_level,
                           u8 *current_mode)
```

Probe a display panel's TCON using the standard VESA eDP backlight interface.

Parameters

struct *drm_dp_aux* *aux

The DP aux device to use for probing

struct *drm_edp_backlight_info* *bl

The *drm_edp_backlight_info* struct to fill out with information on the backlight

u16 driver_pwm_freq_hz

Optional PWM frequency from the driver in hz

const u8 edp_dpcd[EDP_DISPLAY_CTL_CAP_SIZE]

A cached copy of the eDP DPCD

u16 *current_level

Where to store the probed brightness level, if any

u8 *current_mode

Where to store the currently set backlight control mode

Description

Initializes a *drm_edp_backlight_info* struct by probing **aux** for it's backlight capabilities, along with also probing the current and maximum supported brightness levels.

If **driver_pwm_freq_hz** is non-zero, this will be used as the backlight frequency. Otherwise, the default frequency from the panel is used.

Return

0 on success, negative error code on failure.

```
int drm_panel_dp_aux_backlight(struct drm_panel *panel, struct drm_dp_aux *aux)
    create and use DP AUX backlight
```

Parameters

struct *drm_panel* *panel

DRM panel

struct *drm_dp_aux* *aux

The DP AUX channel to use

Description

Use this function to create and handle backlight if your panel supports backlight control over DP AUX channel using DPCD registers as per VESA's standard backlight control interface.

When the panel is enabled backlight will be enabled after a successful call to `drm_panel_funcs.enable()`

When the panel is disabled backlight will be disabled before the call to `drm_panel_funcs.disable()`.

A typical implementation for a panel driver supporting backlight control over DP AUX will call this function at probe time. Backlight will then be handled transparently without requiring any intervention from the driver.

`drm_panel_dp_aux_backlight()` must be called after the call to `drm_panel_init()`.

Return

0 on success or a negative error code on failure.

5.13 Display Port CEC Helper Functions Reference

These functions take care of supporting the CEC-Tunneling-over-AUX feature of DisplayPort-to-HDMI adapters.

void **drm_dp_cec_irq**(struct *drm_dp_aux* *aux)
 handle CEC interrupt, if any

Parameters

struct *drm_dp_aux* *aux
 DisplayPort AUX channel

Description

Should be called when handling an IRQ_HPD request. If CEC-tunneling-over-AUX is present, then it will check for a CEC_IRQ and handle it accordingly.

void **drm_dp_cec_register_connector**(struct *drm_dp_aux* *aux, struct *drm_connector* *connector)
 register a new connector

Parameters

struct *drm_dp_aux* *aux
 DisplayPort AUX channel

struct *drm_connector* *connector
 drm connector

Description

A new connector was registered with associated CEC adapter name and CEC adapter parent device. After registering the name and parent `drm_dp_cec_set_edid()` is called to check if the connector supports CEC and to register a CEC adapter if that is the case.

void **drm_dp_cec_unregister_connector**(struct *drm_dp_aux* *aux)
 unregister the CEC adapter, if any

Parameters


```
struct drm_dp_aux *aux
    DisplayPort AUX channel
```

5.14 Display Port Dual Mode Adaptor Helper Functions Reference

Helper functions to deal with DP dual mode (aka. DP++) adaptors.

Type 1: Adaptor registers (if any) and the sink DDC bus may be accessed via I2C.

Type 2: Adaptor registers and sink DDC bus can be accessed either via I2C or I2C-over-AUX. Source devices may choose to implement either of these access methods.

```
enum drm_lspcon_mode
```

Constants

```
DRM_LSPCON_MODE_INVALID
    No LSPCON.
```

```
DRM_LSPCON_MODE_LS
    Level shifter mode of LSPCON which drives DP++ to HDMI 1.4 conversion.
```

```
DRM_LSPCON_MODE_PCON
    Protocol converter mode of LSPCON which drives DP++ to HDMI 2.0 active conversion.
```

```
enum drm_dp_dual_mode_type
    Type of the DP dual mode adaptor
```

Constants

```
DRM_DP_DUAL_MODE_NONE
    No DP dual mode adaptor
```

```
DRM_DP_DUAL_MODE_UNKNOWN
    Could be either none or type 1 DVI adaptor
```

```
DRM_DP_DUAL_MODE_TYPE1_DVI
    Type 1 DVI adaptor
```

```
DRM_DP_DUAL_MODE_TYPE1_HDMI
    Type 1 HDMI adaptor
```

```
DRM_DP_DUAL_MODE_TYPE2_DVI
    Type 2 DVI adaptor
```

```
DRM_DP_DUAL_MODE_TYPE2_HDMI
    Type 2 HDMI adaptor
```

```
DRM_DP_DUAL_MODE_LSPCON
    Level shifter / protocol converter
```

```
ssize_t drm_dp_dual_mode_read(struct i2c_adapter *adapter, u8 offset, void *buffer, size_t
    size)
```

Read from the DP dual mode adaptor register(s)

Parameters

```
struct i2c_adapter *adapter
    I2C adapter for the DDC bus
```

u8 offset

register offset

void *buffer

buffer for return data

size_t size

size of the buffer

Description

Reads **size** bytes from the DP dual mode adaptor registers starting at **offset**.

Return

0 on success, negative error code on failure

ssize_t **drm_dp_dual_mode_write**(struct i2c_adapter *adapter, u8 offset, const void *buffer, size_t size)

Write to the DP dual mode adaptor register(s)

Parameters

struct i2c_adapter *adapter

I2C adapter for the DDC bus

u8 offset

register offset

const void *buffer

buffer for write data

size_t size

size of the buffer

Description

Writes **size** bytes to the DP dual mode adaptor registers starting at **offset**.

Return

0 on success, negative error code on failure

enum *drm_dp_dual_mode_type* **drm_dp_dual_mode_detect**(const struct *drm_device* *dev, struct i2c_adapter *adapter)

Identify the DP dual mode adaptor

Parameters

const struct drm_device *dev

drm_device to use

struct i2c_adapter *adapter

I2C adapter for the DDC bus

Description

Attempt to identify the type of the DP dual mode adaptor used.

Note that when the answer is **DRM_DP_DUAL_MODE_UNKNOWN** it's not certain whether we're dealing with a native HDMI port or a type 1 DVI dual mode adaptor. The driver will have to use some other hardware/driver specific mechanism to make that distinction.

Return

The type of the DP dual mode adaptor used

```
int drm_dp_dual_mode_max_tmds_clock(const struct drm_device *dev, enum  
                                   drm_dp_dual_mode_type type, struct i2c_adapter  
                                   *adapter)
```

Max TMDS clock for DP dual mode adaptor

Parameters

```
const struct drm_device *dev  
    drm_device to use
```

```
enum drm_dp_dual_mode_type type  
    DP dual mode adaptor type
```

```
struct i2c_adapter *adapter  
    I2C adapter for the DDC bus
```

Description

Determine the max TMDS clock the adaptor supports based on the type of the dual mode adaptor and the DP_DUAL_MODE_MAX_TMDS_CLOCK register (on type2 adaptors). As some type 1 adaptors have problems with registers (see comments in *drm_dp_dual_mode_detect()*) we don't read the register on those, instead we simply assume a 165 MHz limit based on the specification.

Return

Maximum supported TMDS clock rate for the DP dual mode adaptor in kHz.

```
int drm_dp_dual_mode_get_tmds_output(const struct drm_device *dev, enum  
                                   drm_dp_dual_mode_type type, struct i2c_adapter  
                                   *adapter, bool *enabled)
```

Get the state of the TMDS output buffers in the DP dual mode adaptor

Parameters

```
const struct drm_device *dev  
    drm_device to use
```

```
enum drm_dp_dual_mode_type type  
    DP dual mode adaptor type
```

```
struct i2c_adapter *adapter  
    I2C adapter for the DDC bus
```

```
bool *enabled  
    current state of the TMDS output buffers
```

Description

Get the state of the TMDS output buffers in the adaptor. For type2 adaptors this is queried from the DP_DUAL_MODE_TMDS_OEN register. As some type 1 adaptors have problems with registers (see comments in *drm_dp_dual_mode_detect()*) we don't read the register on those, instead we simply assume that the buffers are always enabled.

Return

0 on success, negative error code on failure

int **drm_dp_dual_mode_set_tmds_output**(const struct *drm_device* *dev, enum *drm_dp_dual_mode_type* type, struct i2c_adapter *adapter, bool enable)

Enable/disable TMDS output buffers in the DP dual mode adaptor

Parameters

const struct **drm_device** *dev
drm_device to use

enum **drm_dp_dual_mode_type** type
DP dual mode adaptor type

struct **i2c_adapter** *adapter
I2C adapter for the DDC bus

bool **enable**
enable (as opposed to disable) the TMDS output buffers

Description

Set the state of the TMDS output buffers in the adaptor. For type2 this is set via the DP_DUAL_MODE_TMDS_OEN register. Type1 adaptors do not support any register writes.

Return

0 on success, negative error code on failure

const char ***drm_dp_get_dual_mode_type_name**(enum *drm_dp_dual_mode_type* type)
Get the name of the DP dual mode adaptor type as a string

Parameters

enum **drm_dp_dual_mode_type** type
DP dual mode adaptor type

Return

String representation of the DP dual mode adaptor type

int **drm_lspcon_get_mode**(const struct *drm_device* *dev, struct i2c_adapter *adapter, enum *drm_lspcon_mode* *mode)

Get LSPCON's current mode of operation by reading offset (0x80, 0x41)

Parameters

const struct **drm_device** *dev
drm_device to use

struct **i2c_adapter** *adapter
I2C-over-aux adapter

enum **drm_lspcon_mode** *mode
current lspcon mode of operation output variable

Return

0 on success, sets the current_mode value to appropriate mode -error on failure

```
int drm_lspcon_set_mode(const struct drm_device *dev, struct i2c_adapter *adapter, enum
                        drm_lspcon_mode mode)
```

Change LSPCON's mode of operation by writing offset (0x80, 0x40)

Parameters

const struct *drm_device* *dev

drm_device to use

struct i2c_adapter *adapter

I2C-over-aux adapter

enum *drm_lspcon_mode* mode

required mode of operation

Return

0 on success, -error on failure/timeout

5.15 Display Port MST Helpers

5.15.1 Overview

These functions contain parts of the DisplayPort 1.2a MultiStream Transport protocol. The helpers contain a topology manager and bandwidth manager. The helpers encapsulate the sending and received of sideband msgs.

Topology refcount overview

The refcounting schemes for *struct drm_dp_mst_branch* and *struct drm_dp_mst_port* are somewhat unusual. Both ports and branch devices have two different kinds of refcounts: topology refcounts, and malloc refcounts.

Topology refcounts are not exposed to drivers, and are handled internally by the DP MST helpers. The helpers use them in order to prevent the in-memory topology state from being changed in the middle of critical operations like changing the internal state of payload allocations. This means each branch and port will be considered to be connected to the rest of the topology until its topology refcount reaches zero. Additionally, for ports this means that their associated *struct drm_connector* will stay registered with userspace until the port's refcount reaches 0.

Malloc refcount overview

Malloc references are used to keep a *struct drm_dp_mst_port* or *struct drm_dp_mst_branch* allocated even after all of its topology references have been dropped, so that the driver or MST helpers can safely access each branch's last known state before it was disconnected from the topology. When the malloc refcount of a port or branch reaches 0, the memory allocation containing the *struct drm_dp_mst_branch* or *struct drm_dp_mst_port* respectively will be freed.

For *struct drm_dp_mst_branch*, malloc refcounts are not currently exposed to drivers. As of writing this documentation, there are no drivers that have a usecase for accessing *struct*

`drm_dp_mst_branch` outside of the MST helpers. Exposing this API to drivers in a race-free manner would take more tweaking of the refcounting scheme, however patches are welcome provided there is a legitimate driver usecase for this.

Refcount relationships in a topology

Let's take a look at why the relationship between topology and malloc refcounts is designed the way it is.

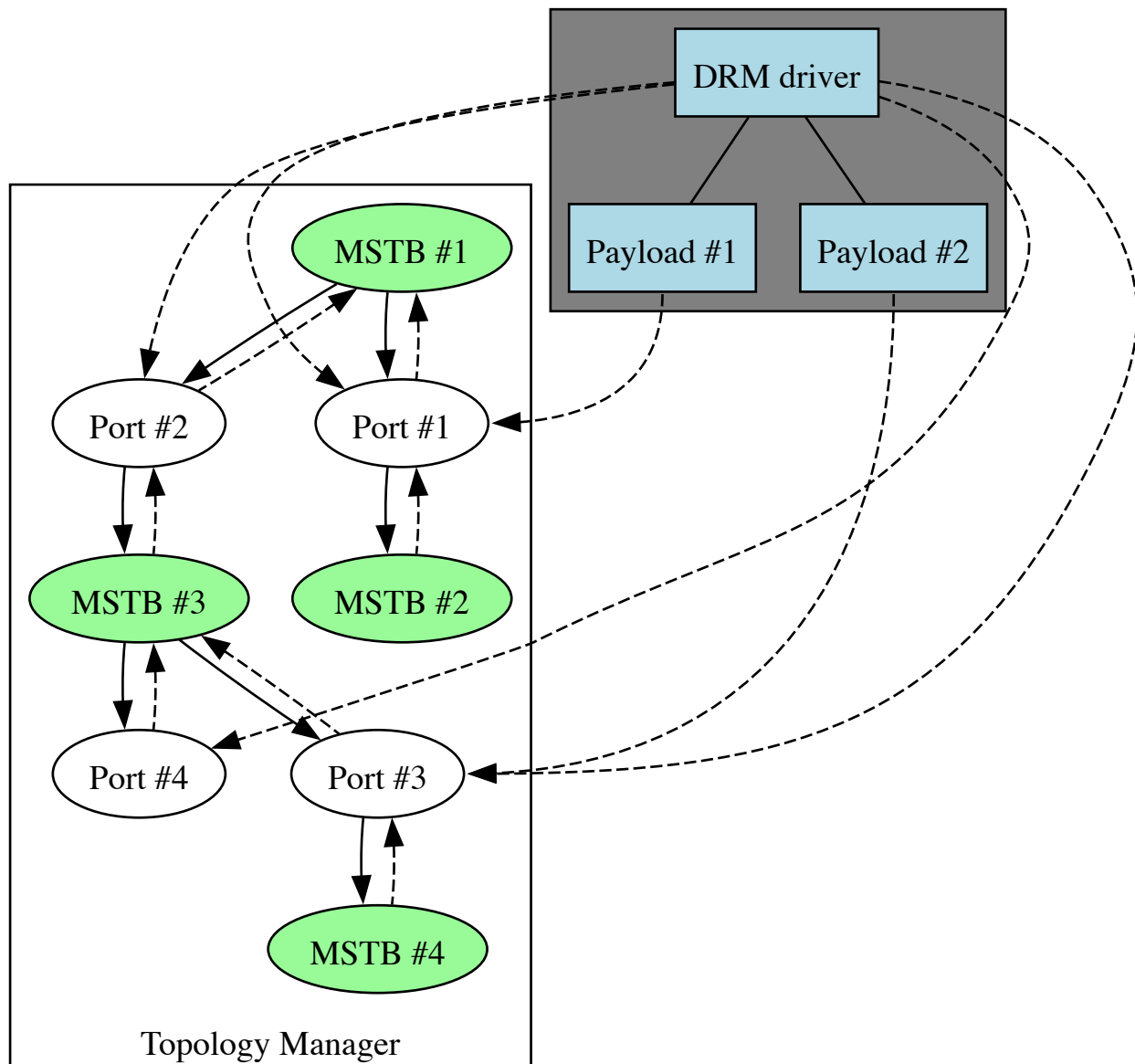


Fig. 1: An example of topology and malloc refs in a DP MST topology with two active payloads. Topology refcount increments are indicated by solid lines, and malloc refcount increments are indicated by dashed lines. Each starts from the branch which incremented the refcount, and ends at the branch to which the refcount belongs to, i.e. the arrow points the same way as the C pointers used to reference a structure.

As you can see in the above figure, every branch increments the topology refcount of its children,

and increments the malloc refcount of its parent. Additionally, every payload increments the malloc refcount of its assigned port by 1.

So, what would happen if MSTB #3 from the above figure was unplugged from the system, but the driver hadn't yet removed payload #2 from port #3? The topology would start to look like the figure below.

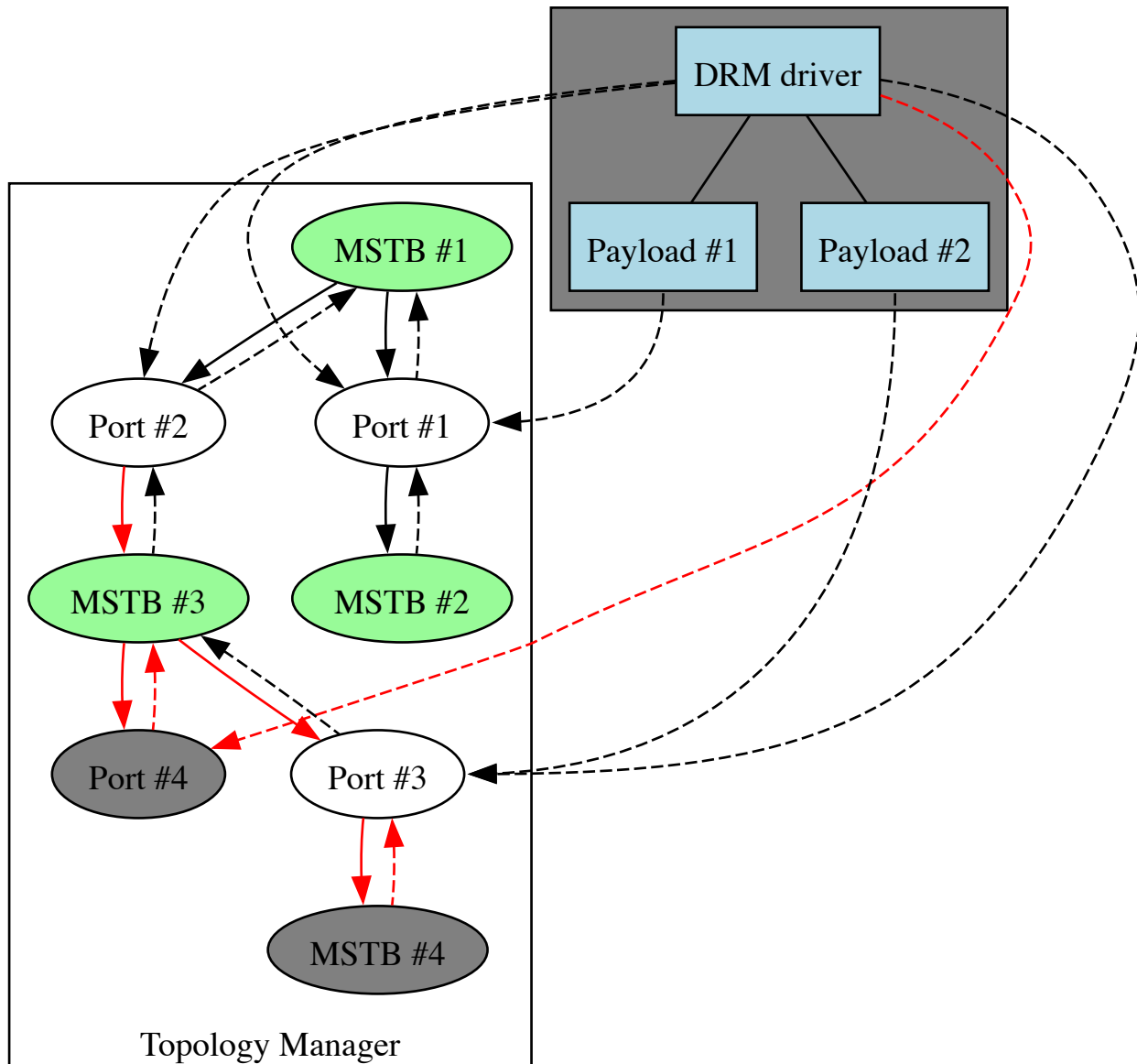
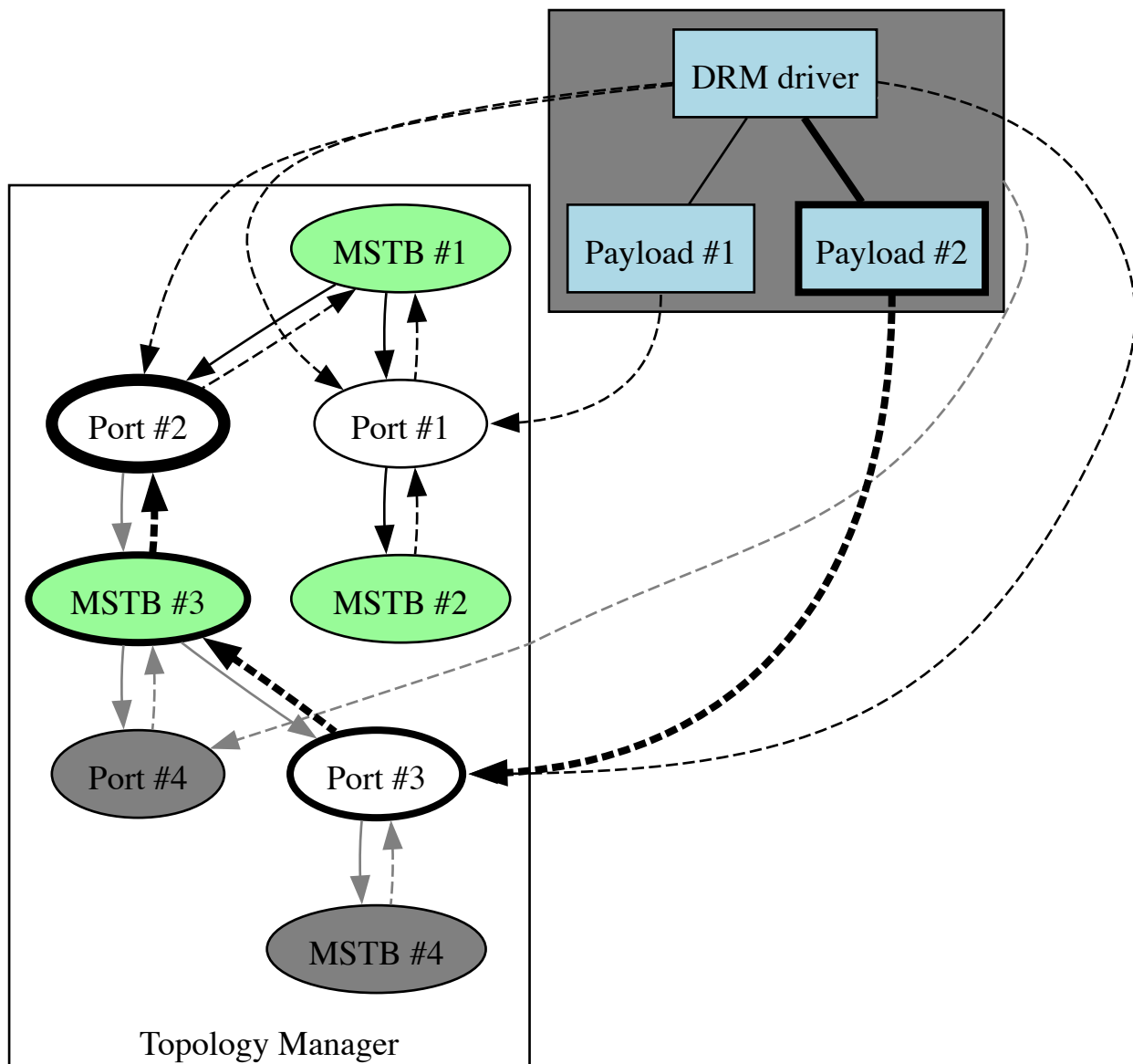


Fig. 2: Ports and branch devices which have been released from memory are colored grey, and references which have been removed are colored red.

Whenever a port or branch device's topology refcount reaches zero, it will decrement the topology refcounts of all its children, the malloc refcount of its parent, and finally its own malloc refcount. For MSTB #4 and port #4, this means they both have been disconnected from the topology and freed from memory. But, because payload #2 is still holding a reference to port #3, port #3 is removed from the topology but its `struct drm_dp_mst_port` is still accessible from memory. This also means port #3 has not yet decremented the malloc refcount of MSTB #3, so its `struct drm_dp_mst_branch` will also stay allocated in memory until port #3's malloc refcount reaches 0.

This relationship is necessary because in order to release payload #2, we need to be able to figure out the last relative of port #3 that's still connected to the topology. In this case, we would travel up the topology as shown below.



And finally, remove payload #2 by communicating with port #2 through sideband transactions.

5.15.2 Functions Reference

struct **drm_dp_mst_port**
MST port

Definition:

```
struct drm_dp_mst_port {
    struct kref topology_kref;
    struct kref malloc_kref;
```



```

#ifdef CONFIG_DRM_DEBUG_DP_MST_TOPOLOGY_REFS;
    struct drm_dp_mst_topology_ref_history topology_ref_history;
#endif;
    u8 port_num;
    bool input;
    bool mcs;
    bool ddps;
    u8 pdt;
    bool ldps;
    u8 dpcd_rev;
    u8 num_sdp_streams;
    u8 num_sdp_stream_sinks;
    uint16_t full_pbn;
    struct list_head next;
    struct drm_dp_mst_branch *mstb;
    struct drm_dp_aux aux;
    struct drm_dp_aux *passthrough_aux;
    struct drm_dp_mst_branch *parent;
    struct drm_connector *connector;
    struct drm_dp_mst_topology_mgr *mgr;
    const struct drm_edid *cached_edid;
    bool fec_capable;
};

```

Members

topology_kref

refcount for this port's lifetime in the topology, only the DP MST helpers should need to touch this

malloc_kref

refcount for the memory allocation containing this structure. See [drm_dp_mst_get_port_malloc\(\)](#) and [drm_dp_mst_put_port_malloc\(\)](#).

topology_ref_history

A history of each topology reference/dereference. See [CONFIG_DRM_DEBUG_DP_MST_TOPOLOGY_REFS](#).

port_num

port number

input

if this port is an input port. Protected by [drm_dp_mst_topology_mgr.base.lock](#).

mcs

message capability status - DP 1.2 spec. Protected by [drm_dp_mst_topology_mgr.base.lock](#).

ddps

DisplayPort Device Plug Status - DP 1.2. Protected by [drm_dp_mst_topology_mgr.base.lock](#).

pdt

Peer Device Type. Protected by [drm_dp_mst_topology_mgr.base.lock](#).

ldps

Legacy Device Plug Status. Protected by *drm_dp_mst_topology_mgr.base.lock*.

dpcd_rev

DPCD revision of device on this port. Protected by *drm_dp_mst_topology_mgr.base.lock*.

num_sdp_streams

Number of simultaneous streams. Protected by *drm_dp_mst_topology_mgr.base.lock*.

num_sdp_stream_sinks

Number of stream sinks. Protected by *drm_dp_mst_topology_mgr.base.lock*.

full_pbn

Max possible bandwidth for this port. Protected by *drm_dp_mst_topology_mgr.base.lock*.

next

link to next port on this branch device

mstb

the branch device connected to this port, if there is one. This should be considered protected for reading by *drm_dp_mst_topology_mgr.lock*. There are two exceptions to this: *drm_dp_mst_topology_mgr.up_req_work* and *drm_dp_mst_topology_mgr.work*, which do not grab *drm_dp_mst_topology_mgr.lock* during reads but are the only updaters of this list and are protected from writing concurrently by *drm_dp_mst_topology_mgr.probe_lock*.

aux

i2c aux transport to talk to device connected to this port, protected by *drm_dp_mst_topology_mgr.base.lock*.

passthrough_aux

parent aux to which DSC pass-through requests should be sent, only set if DSC pass-through is possible.

parent

branch device parent of this port

connector

DRM connector this port is connected to. Protected by *drm_dp_mst_topology_mgr.base.lock*.

mgr

topology manager this port lives under.

cached_edid

for DP logical ports - make tiling work by ensuring that the EDID for all connectors is read immediately.

fec_capable

bool indicating if FEC can be supported up to that point in the MST topology.

Description

This structure represents an MST port endpoint on a device somewhere in the MST topology.

struct `drm_dp_mst_branch`

MST branch device.

Definition:

```

struct drm_dp_mst_branch {
    struct kref topology_kref;
    struct kref malloc_kref;
#ifdef IS_ENABLED(CONFIG_DRM_DEBUG_DP_MST_TOPOLOGY_REFS);
    struct drm_dp_mst_topology_ref_history topology_ref_history;
#endif;
    struct list_head destroy_next;
    u8 rad[8];
    u8 lct;
    int num_ports;
    struct list_head ports;
    struct drm_dp_mst_port *port_parent;
    struct drm_dp_mst_topology_mgr *mgr;
    bool link_address_sent;
    u8 guid[16];
};

```

Members

topology_kref

refcount for this branch device's lifetime in the topology, only the DP MST helpers should need to touch this

malloc_kref

refcount for the memory allocation containing this structure. See [drm_dp_mst_get_mstb_malloc\(\)](#) and [drm_dp_mst_put_mstb_malloc\(\)](#).

topology_ref_history

A history of each topology reference/dereference. See [CONFIG_DRM_DEBUG_DP_MST_TOPOLOGY_REFS](#).

destroy_next

linked-list entry used by [drm_dp_delayed_destroy_work\(\)](#)

rad

Relative Address to talk to this branch device.

lct

Link count total to talk to this branch device.

num_ports

number of ports on the branch.

ports

the list of ports on this branch device. This should be considered protected for reading by [drm_dp_mst_topology_mgr.lock](#). There are two exceptions to this: [drm_dp_mst_topology_mgr.up_req_work](#) and [drm_dp_mst_topology_mgr.work](#), which do not grab [drm_dp_mst_topology_mgr.lock](#) during reads but are the only updaters of this list and are protected from updating the list concurrently by [drm_dp_mst_topology_mgr.probe_lock](#)

port_parent

pointer to the port parent, NULL if toplevel.

mgr

topology manager for this branch device.

link_address_sent

if a link address message has been sent to this device yet.

guid

guid for DP 1.2 branch device. port under this branch can be identified by port #.

Description

This structure represents an MST branch device, there is one primary branch device at the root, along with any other branches connected to downstream port of parent branches.

struct `drm_dp_mst_atomic_payload`

Atomic state struct for an MST payload

Definition:

```
struct drm_dp_mst_atomic_payload {
    struct drm_dp_mst_port *port;
    s8 vc_start_slot;
    u8 vcpi;
    int time_slots;
    int pbn;
    bool delete : 1;
    bool dsc_enabled : 1;
    struct list_head next;
};
```

Members

port

The MST port assigned to this payload

vc_start_slot

The time slot that this payload starts on. Because payload start slots can't be determined ahead of time, the contents of this value are UNDEFINED at atomic check time. This shouldn't usually matter, as the start slot should never be relevant for atomic state computations.

Since this value is determined at commit time instead of check time, this value is protected by the MST helpers ensuring that async commits operating on the given topology never run in parallel. In the event that a driver does need to read this value (e.g. to inform hardware of the starting timeslot for a payload), the driver may either:

- Read this field during the atomic commit after [`drm_dp_mst_atomic_wait_for_dependencies`](#) has been called, which will ensure the previous MST states payload start slots have been copied over to the new state. Note that a new start slot won't be assigned/removed from this payload until [`drm_dp_add_payload_part1\(\)`](#)/[`drm_dp_remove_payload\(\)`](#) have been called.
- Acquire the MST modesetting lock, and then wait for any pending MST-related commits to get committed to hardware by calling [`drm_crtc_commit_wait\(\)`](#) on each of the [`drm_crtc_commit`](#) structs in [`drm_dp_mst_topology_state.commit_deps`](#).

If neither of the two above solutions suffice (e.g. the driver needs to read the start slot in the middle of an atomic commit without waiting for some reason), then drivers should cache this value themselves after changing payloads.

vcpi

The Virtual Channel Payload Identifier

time_slots

The number of timeslots allocated to this payload from the source DP Tx to the immediate downstream DP Rx

pbn

The payload bandwidth for this payload

delete

Whether or not we intend to delete this payload during this atomic commit

dsc_enabled

Whether or not this payload has DSC enabled

next

The list node for this payload

Description

The primary atomic state structure for a given MST payload. Stores information like current bandwidth allocation, intended action for this payload, etc.

struct `drm_dp_mst_topology_state`

DisplayPort MST topology atomic state

Definition:

```
struct drm_dp_mst_topology_state {
    struct drm_private_state base;
    struct drm_dp_mst_topology_mgr *mgr;
    u32 pending_crtc_mask;
    struct drm_crtc_commit **commit_deps;
    size_t num_commit_deps;
    u32 payload_mask;
    struct list_head payloads;
    u8 total_avail_slots;
    u8 start_slot;
    int pbn_div;
};
```

Members**base**

Base private state for atomic

mgr

The topology manager

pending_crtc_mask

A bitmask of all CRTC's this topology state touches, drivers may modify this to add additional dependencies if needed.

commit_deps

A list of all CRTC commits affecting this topology, this field isn't populated until [`drm_dp_mst_atomic_wait_for_dependencies\(\)`](#) is called.

num_commit_deps

The number of CRTC commits in **commit_deps**

payload_mask

A bitmask of allocated VCPIs, used for VCPI assignments

payloads

The list of payloads being created/destroyed in this state

total_avail_slots

The total number of slots this topology can handle (63 or 64)

start_slot

The first usable time slot in this topology (1 or 0)

pbn_div

The current PBN divisor for this topology. The driver is expected to fill this out itself.

Description

This struct represents the atomic state of the toplevel DisplayPort MST manager

struct drm_dp_mst_topology_mgr

DisplayPort MST manager

Definition:

```
struct drm_dp_mst_topology_mgr {
    struct drm_private_obj base;
    struct drm_device *dev;
    const struct drm_dp_mst_topology_cbs *cbs;
    int max_dpcd_transaction_bytes;
    struct drm_dp_aux *aux;
    int max_payloads;
    int conn_base_id;
    struct drm_dp_sideband_msg_rx up_req_recv;
    struct drm_dp_sideband_msg_rx down_rep_recv;
    struct mutex lock;
    struct mutex probe_lock;
    bool mst_state : 1;
    bool payload_id_table_cleared : 1;
    u8 payload_count;
    u8 next_start_slot;
    struct drm_dp_mst_branch *mst_primary;
    u8 dpcd[DP_RECEIVER_CAP_SIZE];
    u8 sink_count;
    const struct drm_private_state_funcs *funcs;
    struct mutex qlock;
    struct list_head tx_msg_downq;
    wait_queue_head_t tx_waitq;
    struct work_struct work;
    struct work_struct tx_work;
    struct list_head destroy_port_list;
    struct list_head destroy_branch_device_list;
    struct mutex delayed_destroy_lock;
    struct workqueue_struct *delayed_destroy_wq;
```

```

struct work_struct delayed_destroy_work;
struct list_head up_req_list;
struct mutex up_req_lock;
struct work_struct up_req_work;
#if IS_ENABLED(CONFIG_DRM_DEBUG_DP_MST_TOPOLOGY_REFS);
struct mutex topology_ref_history_lock;
#endif;
};

```

Members

base

Base private object for atomic

dev

device pointer for adding i2c devices etc.

cbs

callbacks for connector addition and destruction.

max_dpcd_transaction_bytes

maximum number of bytes to read/write in one go.

aux

AUX channel for the DP MST connector this topology mgr is controlling.

max_payloads

maximum number of payloads the GPU can generate.

conn_base_id

DRM connector ID this mgr is connected to. Only used to build the MST connector path value.

up_req_recv

Message receiver state for up requests.

down_rep_recv

Message receiver state for replies to down requests.

lock

protects **mst_state**, **mst_primary**, **dpcd**, and **payload_id_table_cleared**.

probe_lock

Prevents **work** and **up_req_work**, the only writers of *drm_dp_mst_port.mstb* and *drm_dp_mst_branch.ports*, from racing while they update the topology.

mst_state

If this manager is enabled for an MST capable port. False if no MST sink/branch devices is connected.

payload_id_table_cleared

Whether or not we've cleared the payload ID table for **mst_primary**. Protected by **lock**.

payload_count

The number of currently active payloads in hardware. This value is only intended to be used internally by MST helpers for payload tracking, and is only safe to read/write from the atomic commit (not check) context.

next_start_slot

The starting timeslot to use for new VC payloads. This value is used internally by MST helpers for payload tracking, and is only safe to read/write from the atomic commit (not check) context.

mst_primary

Pointer to the primary/first branch device.

dpcd

Cache of DPCD for primary port.

sink_count

Sink count from DEVICE_SERVICE_IRQ_VECTOR_ESI0.

funcs

Atomic helper callbacks

qlock

protects **tx_msg_downq** and `drm_dp_sideband_msg_tx.state`

tx_msg_downq

List of pending down requests

tx_waitq

Wait to queue stall for the tx worker.

work

Probe work.

tx_work

Sideband transmit worker. This can nest within the main **work** worker for each transaction **work** launches.

destroy_port_list

List of to be destroyed connectors.

destroy_branch_device_list

List of to be destroyed branch devices.

delayed_destroy_lock

Protects **destroy_port_list** and **destroy_branch_device_list**.

delayed_destroy_wq

Workqueue used for `delayed_destroy_work` items. A dedicated WQ makes it possible to drain any requeued work items on it.

delayed_destroy_work

Work item to destroy MST port and branch devices, needed to avoid locking inversion.

up_req_list

List of pending up requests from the topology that need to be processed, in chronological order.

up_req_lock

Protects **up_req_list**

up_req_work

Work item to process up requests received from the topology. Needed to avoid blocking hotplug handling and sideband transmissions.

topology_ref_history_lock

protects `drm_dp_mst_port.topology_ref_history` and `drm_dp_mst_branch.topology_ref_history`.

Description

This struct represents the toplevel displayport MST topology manager. There should be one instance of this for every MST capable DP connector on the GPU.

```
bool __drm_dp_mst_state_iter_get(struct drm_atomic_state *state, struct
                                drm_dp_mst_topology_mgr **mgr, struct
                                drm_dp_mst_topology_state **old_state, struct
                                drm_dp_mst_topology_state **new_state, int i)
```

private atomic state iterator function for macro-internal use

Parameters

struct drm_atomic_state *state

`struct drm_atomic_state` pointer

struct drm_dp_mst_topology_mgr **mgr

pointer to the `struct drm_dp_mst_topology_mgr` iteration cursor

struct drm_dp_mst_topology_state **old_state

optional pointer to the old `struct drm_dp_mst_topology_state` iteration cursor

struct drm_dp_mst_topology_state **new_state

optional pointer to the new `struct drm_dp_mst_topology_state` iteration cursor

int i

int iteration cursor, for macro-internal use

Description

Used by `for_each_oldnew_mst_mgr_in_state()`, `for_each_old_mst_mgr_in_state()`, and `for_each_new_mst_mgr_in_state()`. Don't call this directly.

Return

True if the current `struct drm_private_obj` is a `struct drm_dp_mst_topology_mgr`, false otherwise.

for_each_oldnew_mst_mgr_in_state

```
for_each_oldnew_mst_mgr_in_state (__state, mgr, old_state, new_state, __i)
```

iterate over all DP MST topology managers in an atomic update

Parameters

__state

`struct drm_atomic_state` pointer

mgr

`struct drm_dp_mst_topology_mgr` iteration cursor

old_state

`struct drm_dp_mst_topology_state` iteration cursor for the old state

new_state

`struct drm_dp_mst_topology_state` iteration cursor for the new state

__i
int iteration cursor, for macro-internal use

Description

This iterates over all DRM DP MST topology managers in an atomic update, tracking both old and new state. This is useful in places where the state delta needs to be considered, for example in atomic check functions.

for_each_old_mst_mgr_in_state

`for_each_old_mst_mgr_in_state (__state, mgr, old_state, __i)`
iterate over all DP MST topology managers in an atomic update

Parameters

__state
struct drm_atomic_state pointer

mgr
struct drm_dp_mst_topology_mgr iteration cursor

old_state
struct drm_dp_mst_topology_state iteration cursor for the old state

__i
int iteration cursor, for macro-internal use

Description

This iterates over all DRM DP MST topology managers in an atomic update, tracking only the old state. This is useful in disable functions, where we need the old state the hardware is still in.

for_each_new_mst_mgr_in_state

`for_each_new_mst_mgr_in_state (__state, mgr, new_state, __i)`
iterate over all DP MST topology managers in an atomic update

Parameters

__state
struct drm_atomic_state pointer

mgr
struct drm_dp_mst_topology_mgr iteration cursor

new_state
struct drm_dp_mst_topology_state iteration cursor for the new state

__i
int iteration cursor, for macro-internal use

Description

This iterates over all DRM DP MST topology managers in an atomic update, tracking only the new state. This is useful in enable functions, where we need the new state the hardware should be in when the atomic commit operation has completed.

void **drm_dp_mst_get_port_malloc**(struct *drm_dp_mst_port* *port)

Increment the malloc refcount of an MST port

Parameters

struct drm_dp_mst_port *port

The *struct drm_dp_mst_port* to increment the malloc refcount of

Description

Increments *drm_dp_mst_port.malloc_kref*. When *drm_dp_mst_port.malloc_kref* reaches 0, the memory allocation for **port** will be released and **port** may no longer be used.

Because **port** could potentially be freed at any time by the DP MST helpers if *drm_dp_mst_port.malloc_kref* reaches 0, including during a call to this function, drivers that which to make use of *struct drm_dp_mst_port* should ensure that they grab at least one main malloc reference to their MST ports in *drm_dp_mst_topology_cbs.add_connector*. This callback is called before there is any chance for *drm_dp_mst_port.malloc_kref* to reach 0.

See also: *drm_dp_mst_put_port_malloc()*

void **drm_dp_mst_put_port_malloc**(struct *drm_dp_mst_port* *port)

Decrement the malloc refcount of an MST port

Parameters

struct drm_dp_mst_port *port

The *struct drm_dp_mst_port* to decrement the malloc refcount of

Description

Decrements *drm_dp_mst_port.malloc_kref*. When *drm_dp_mst_port.malloc_kref* reaches 0, the memory allocation for **port** will be released and **port** may no longer be used.

See also: *drm_dp_mst_get_port_malloc()*

int **drm_dp_mst_connector_late_register**(struct *drm_connector* *connector, struct *drm_dp_mst_port* *port)

Late MST connector registration

Parameters

struct drm_connector *connector

The MST connector

struct drm_dp_mst_port *port

The MST port for this connector

Description

Helper to register the remote aux device for this MST port. Drivers should call this from their mst connector's *late_register* hook to enable MST aux devices.

Return

0 on success, negative error code on failure.

void **drm_dp_mst_connector_early_unregister**(struct *drm_connector* *connector, struct *drm_dp_mst_port* *port)

Early MST connector unregistration

Parameters

struct drm_connector *connector

The MST connector

struct drm_dp_mst_port *port

The MST port for this connector

Description

Helper to unregister the remote aux device for this MST port, registered by *drm_dp_mst_connector_late_register()*. Drivers should call this from their mst connector's early_unregister hook.

int **drm_dp_add_payload_part1**(struct *drm_dp_mst_topology_mgr* *mgr, struct *drm_dp_mst_topology_state* *mst_state, struct *drm_dp_mst_atomic_payload* *payload)

Execute payload update part 1

Parameters

struct drm_dp_mst_topology_mgr *mgr

Manager to use.

struct drm_dp_mst_topology_state *mst_state

The MST atomic state

struct drm_dp_mst_atomic_payload *payload

The payload to write

Description

Determines the starting time slot for the given payload, and programs the VCPI for this payload into hardware. After calling this, the driver should generate ACT and payload packets.

Return

0 on success, error code on failure. In the event that this fails, **payload.vc_start_slot** will also be set to -1.

void **drm_dp_remove_payload**(struct *drm_dp_mst_topology_mgr* *mgr, struct *drm_dp_mst_topology_state* *mst_state, const struct *drm_dp_mst_atomic_payload* *old_payload, struct *drm_dp_mst_atomic_payload* *new_payload)

Remove an MST payload

Parameters

struct drm_dp_mst_topology_mgr *mgr

Manager to use.

struct drm_dp_mst_topology_state *mst_state

The MST atomic state

const struct drm_dp_mst_atomic_payload *old_payload

The payload with its old state

struct drm_dp_mst_atomic_payload *new_payload

The payload to write

Description

Removes a payload from an MST topology if it was successfully assigned a start slot. Also updates the starting time slots of all other payloads which would have been shifted towards the start of the VC table as a result. After calling this, the driver should generate ACT and payload packets.

```
int drm_dp_add_payload_part2(struct drm_dp_mst_topology_mgr *mgr, struct
                           drm_atomic_state *state, struct drm_dp_mst_atomic_payload
                           *payload)
```

Execute payload update part 2

Parameters

struct *drm_dp_mst_topology_mgr* *mgr
Manager to use.

struct *drm_atomic_state* *state
The global atomic state

struct *drm_dp_mst_atomic_payload* *payload
The payload to update

Description

If **payload** was successfully assigned a starting time slot by *drm_dp_add_payload_part1()*, this function will send the sideband messages to finish allocating this payload.

Return

0 on success, negative error code on failure.

```
int drm_dp_get_vc_payload_bw(const struct drm_dp_mst_topology_mgr *mgr, int link_rate,
                             int link_lane_count)
```

get the VC payload BW for an MST link

Parameters

const struct *drm_dp_mst_topology_mgr* *mgr
The *drm_dp_mst_topology_mgr* to use

int link_rate
link rate in 10kbits/s units

int link_lane_count
lane count

Description

Calculate the total bandwidth of a MultiStream Transport link. The returned value is in units of PBNs/(timeslots/1 MTP). This value can be used to convert the number of PBNs required for a given stream to the number of timeslots this stream requires in each MTP.

```
bool drm_dp_read_mst_cap(struct drm_dp_aux *aux, const u8
                        dpcd[DP_RECEIVER_CAP_SIZE])
```

check whether or not a sink supports MST

Parameters

struct drm_dp_aux *aux

The DP AUX channel to use

const u8 dpcd[DP_RECEIVER_CAP_SIZE]

A cached copy of the DPCD capabilities for this sink

Return

True if the sink supports MST, false otherwise

int drm_dp_mst_topology_mgr_set_mst(struct *drm_dp_mst_topology_mgr* *mgr, bool mst_state)

Set the MST state for a topology manager

Parameters

struct drm_dp_mst_topology_mgr *mgr

manager to set state for

bool mst_state

true to enable MST on this connector - false to disable.

Description

This is called by the driver when it detects an MST capable device plugged into a DP MST capable port, or when a DP MST capable device is unplugged.

void drm_dp_mst_topology_mgr_suspend(struct *drm_dp_mst_topology_mgr* *mgr)

suspend the MST manager

Parameters

struct drm_dp_mst_topology_mgr *mgr

manager to suspend

Description

This function tells the MST device that we can't handle UP messages anymore. This should stop it from sending any since we are suspended.

int drm_dp_mst_topology_mgr_resume(struct *drm_dp_mst_topology_mgr* *mgr, bool sync)

resume the MST manager

Parameters

struct drm_dp_mst_topology_mgr *mgr

manager to resume

bool sync

whether or not to perform topology reprobing synchronously

Description

This will fetch DPCD and see if the device is still there, if it is, it will rewrite the MSTM control bits, and return.

If the device fails this returns -1, and the driver should do a full MST reprobe, in case we were undocked.

During system resume (where it is assumed that the driver will be calling *drm_atomic_helper_resume()*) this function should be called beforehand with **sync** set to true. In contexts like runtime resume where the driver is not expected to be calling

drm_atomic_helper_resume(), this function should be called with **sync** set to false in order to avoid deadlocking.

Return

-1 if the MST topology was removed while we were suspended, 0 otherwise.

int **drm_dp_mst_hpd_irq_handle_event**(struct *drm_dp_mst_topology_mgr* *mgr, const u8 *esi, u8 *ack, bool *handled)

MST hotplug IRQ handle MST event

Parameters

struct **drm_dp_mst_topology_mgr** *mgr
manager to notify irq for.

const u8 *esi
4 bytes from SINK_COUNT_ESI

u8 *ack
4 bytes used to ack events starting from SINK_COUNT_ESI

bool *handled
whether the hpd interrupt was consumed or not

Description

This should be called from the driver when it detects a HPD IRQ, along with the value of the DEVICE_SERVICE_IRQ_VECTOR_ESI0. The topology manager will process the sideband messages received as indicated in the DEVICE_SERVICE_IRQ_VECTOR_ESI0 and set the corresponding flags that Driver has to ack the DP receiver later.

Note that driver shall also call *drm_dp_mst_hpd_irq_send_new_request()* if the 'handled' is set after calling this function, to try to kick off a new request in the queue if the previous message transaction is completed.

See also: *drm_dp_mst_hpd_irq_send_new_request()*

void **drm_dp_mst_hpd_irq_send_new_request**(struct *drm_dp_mst_topology_mgr* *mgr)
MST hotplug IRQ kick off new request

Parameters

struct **drm_dp_mst_topology_mgr** *mgr
manager to notify irq for.

Description

This should be called from the driver when mst irq event is handled and acked. Note that new down request should only be sent when previous message transaction is completed. Source is not supposed to generate interleaved message transactions.

int **drm_dp_mst_detect_port**(struct *drm_connector* *connector, struct *drm_modeset_acquire_ctx* *ctx, struct *drm_dp_mst_topology_mgr* *mgr, struct *drm_dp_mst_port* *port)

get connection status for an MST port

Parameters

struct drm_connector *connector

DRM connector for this port

struct drm_modeset_acquire_ctx *ctx

The acquisition context to use for grabbing locks

struct drm_dp_mst_topology_mgr *mgr

manager for this port

struct drm_dp_mst_port *port

pointer to a port

Description

This returns the current connection state for a port.

```
const struct drm_edid *drm_dp_mst_edid_read(struct drm_connector *connector, struct  
                                             drm_dp_mst_topology_mgr *mgr, struct  
                                             drm_dp_mst_port *port)
```

get EDID for an MST port

Parameters

struct drm_connector *connector

toplevel connector to get EDID for

struct drm_dp_mst_topology_mgr *mgr

manager for this port

struct drm_dp_mst_port *port

unverified pointer to a port.

Description

This returns an EDID for the port connected to a connector, It validates the pointer still exists so the caller doesn't require a reference.

```
struct edid *drm_dp_mst_get_edid(struct drm_connector *connector, struct  
                                 drm_dp_mst_topology_mgr *mgr, struct drm_dp_mst_port  
                                 *port)
```

get EDID for an MST port

Parameters

struct drm_connector *connector

toplevel connector to get EDID for

struct drm_dp_mst_topology_mgr *mgr

manager for this port

struct drm_dp_mst_port *port

unverified pointer to a port.

Description

This function is deprecated; please use *drm_dp_mst_edid_read()* instead.

This returns an EDID for the port connected to a connector, It validates the pointer still exists so the caller doesn't require a reference.


```
int drm_dp_atomic_find_time_slots(struct drm_atomic_state *state, struct  
                                drm_dp_mst_topology_mgr *mgr, struct  
                                drm_dp_mst_port *port, int pbn)
```

Find and add time slots to the state

Parameters

struct *drm_atomic_state* *state
global atomic state

struct *drm_dp_mst_topology_mgr* *mgr
MST topology manager for the port

struct *drm_dp_mst_port* *port
port to find time slots for

int pbn
bandwidth required for the mode in PBN

Description

Allocates time slots to **port**, replacing any previous time slot allocations it may have had. Any atomic drivers which support MST must call this function in their *drm_encoder_helper_funcs.atomic_check()* callback unconditionally to change the current time slot allocation for the new state, and ensure the MST atomic state is added whenever the state of payloads in the topology changes.

Allocations set by this function are not checked against the bandwidth restraints of **mgr** until the driver calls *drm_dp_mst_atomic_check()*.

Additionally, it is OK to call this function multiple times on the same **port** as needed. It is not OK however, to call this function and *drm_dp_atomic_release_time_slots()* in the same atomic check phase.

See also: *drm_dp_atomic_release_time_slots()* *drm_dp_mst_atomic_check()*

Return

Total slots in the atomic state assigned for this port, or a negative error code if the port no longer exists

```
int drm_dp_atomic_release_time_slots(struct drm_atomic_state *state, struct  
                                   drm_dp_mst_topology_mgr *mgr, struct  
                                   drm_dp_mst_port *port)
```

Release allocated time slots

Parameters

struct *drm_atomic_state* *state
global atomic state

struct *drm_dp_mst_topology_mgr* *mgr
MST topology manager for the port

struct *drm_dp_mst_port* *port
The port to release the time slots from

Description

Releases any time slots that have been allocated to a port in the atomic state. Any atomic drivers which support MST must call this function unconditionally in their `drm_connector_helper_funcs.atomic_check()` callback. This helper will check whether time slots would be released by the new state and respond accordingly, along with ensuring the MST state is always added to the atomic state whenever a new state would modify the state of payloads on the topology.

It is OK to call this even if **port** has been removed from the system. Additionally, it is OK to call this function multiple times on the same **port** as needed. It is not OK however, to call this function and `drm_dp_atomic_find_time_slots()` on the same **port** in a single atomic check phase.

See also: `drm_dp_atomic_find_time_slots()` `drm_dp_mst_atomic_check()`

Return

0 on success, negative error code otherwise

int **drm_dp_mst_atomic_setup_commit**(struct `drm_atomic_state` *state)
setup_commit hook for MST helpers

Parameters

struct drm_atomic_state *state
global atomic state

Description

This function saves all of the `drm_crtc_commit` structs in an atomic state that touch any CRTC's currently assigned to an MST topology. Drivers must call this hook from their `drm_mode_config_helper_funcs.atomic_commit_setup` hook.

Return

0 if all CRTC commits were retrieved successfully, negative error code otherwise

void **drm_dp_mst_atomic_wait_for_dependencies**(struct `drm_atomic_state` *state)
Wait for all pending commits on MST topologies, prepare new MST state for commit

Parameters

struct drm_atomic_state *state
global atomic state

Description

Goes through any MST topologies in this atomic state, and waits for any pending commits which touched CRTC's that were/are on an MST topology to be programmed to hardware and flipped to before returning. This is to prevent multiple non-blocking commits affecting an MST topology from racing with each other by forcing them to be executed sequentially in situations where the only resources the modeset objects in these commits share are an MST topology.

This function also prepares the new MST state for commit by performing some state preparation which can't be done until this point, such as reading back the final VC start slots (which are determined at commit-time) from the previous state.

All MST drivers must call this function after calling `drm_atomic_helper_wait_for_dependencies()`, or whatever their equivalent of that is.

```
int drm_dp_mst_root_conn_atomic_check(struct drm_connector_state *new_conn_state,  
                                       struct drm_dp_mst_topology_mgr *mgr)
```

Serialize CRTC commits on MST-capable connectors operating in SST mode

Parameters

struct *drm_connector_state* *new_conn_state
The new connector state of the *drm_connector*

struct *drm_dp_mst_topology_mgr* *mgr
The MST topology manager for the *drm_connector*

Description

Since MST uses fake *drm_encoder* structs, the generic atomic modesetting code isn't able to serialize non-blocking commits happening on the real DP connector of an MST topology switching into/away from MST mode - as the CRTC on the real DP connector and the CRTCs on the connector's MST topology will never share the same *drm_encoder*.

This function takes care of this serialization issue, by checking a root MST connector's atomic state to determine if it is about to have a modeset - and then pulling in the MST topology state if so, along with adding any relevant CRTCs to *drm_dp_mst_topology_state.pending_crtc_mask*.

Drivers implementing MST must call this function from the *drm_connector_helper_funcs.atomic_check* hook of any physical DP *drm_connector* capable of driving MST sinks.

Return

0 on success, negative error code otherwise

```
void drm_dp_mst_update_slots(struct drm_dp_mst_topology_state *mst_state, uint8_t  
                             link_encoding_cap)
```

updates the slot info depending on the DP encoding format

Parameters

struct *drm_dp_mst_topology_state* *mst_state
mst_state to update

uint8_t link_encoding_cap
the encoding format on the link

```
int drm_dp_check_act_status(struct drm_dp_mst_topology_mgr *mgr)  
Polls for ACT handled status.
```

Parameters

struct *drm_dp_mst_topology_mgr* *mgr
manager to use

Description

Tries waiting for the MST hub to finish updating it's payload table by polling for the ACT handled bit for up to 3 seconds (yes-some hubs really take that long).

Return

0 if the ACT was handled in time, negative error code on failure.

int **drm_dp_calc_pbn_mode**(int clock, int bpp)

Calculate the PBN for a mode.

Parameters

int **clock**

dot clock

int **bpp**

bpp as .4 binary fixed point

Description

This uses the formula in the spec to calculate the PBN value for a mode.

void **drm_dp_mst_dump_topology**(struct seq_file *m, struct *drm_dp_mst_topology_mgr* *mgr)
dump topology to seq file.

Parameters

struct seq_file *m

seq_file to dump output to

struct *drm_dp_mst_topology_mgr* *mgr

manager to dump current topology for.

Description

helper to dump MST topology to a seq file for debugfs.

int **drm_dp_mst_add_affected_dsc_crtcs**(struct *drm_atomic_state* *state, struct *drm_dp_mst_topology_mgr* *mgr)

Parameters

struct *drm_atomic_state* *state

Pointer to the new *struct drm_dp_mst_topology_state*

struct *drm_dp_mst_topology_mgr* *mgr

MST topology manager

Description

Whenever there is a change in mst topology DSC configuration would have to be recalculated therefore we need to trigger modeset on all affected CRTC's in that topology

See also: *drm_dp_mst_atomic_enable_dsc()*

int **drm_dp_mst_atomic_enable_dsc**(struct *drm_atomic_state* *state, struct *drm_dp_mst_port* *port, int pbn, bool enable)

Set DSC Enable Flag to On/Off

Parameters

struct *drm_atomic_state* *state

Pointer to the new *drm_atomic_state*

struct *drm_dp_mst_port* *port

Pointer to the affected MST Port

int **pbn**

Newly recalculated bw required for link with DSC enabled

bool enable

Boolean flag to enable or disable DSC on the port

Description

This function enables DSC on the given Port by recalculating its vcpi from pbn provided and sets dsc_enable flag to keep track of which ports have DSC enabled

int **drm_dp_mst_atomic_check**(struct *drm_atomic_state* *state)

Check that the new state of an MST topology in an atomic update is valid

Parameters

struct drm_atomic_state *state

Pointer to the new *struct drm_dp_mst_topology_state*

Description

Checks the given topology state for an atomic update to ensure that it's valid. This includes checking whether there's enough bandwidth to support the new timeslot allocations in the atomic update.

Any atomic drivers supporting DP MST must make sure to call this after checking the rest of their state in their *drm_mode_config_funcs.atomic_check()* callback.

See also: *drm_dp_atomic_find_time_slots()* *drm_dp_atomic_release_time_slots()*

0 if the new state is valid, negative error code otherwise.

Return

struct *drm_dp_mst_topology_state* ***drm_atomic_get_mst_topology_state**(struct *drm_atomic_state* *state, struct *drm_dp_mst_topology_mgr* *mgr)

get MST topology state

Parameters

struct drm_atomic_state *state

global atomic state

struct drm_dp_mst_topology_mgr *mgr

MST topology manager, also the private object in this case

Description

This function wraps *drm_atomic_get_priv_obj_state()* passing in the MST atomic state vtable so that the private object state returned is that of a MST topology object.

The MST topology state or error pointer.

Return

struct *drm_dp_mst_topology_state* ***drm_atomic_get_old_mst_topology_state**(struct *drm_atomic_state* *state, struct *drm_dp_mst_topology_mgr* *mgr)

get old MST topology state in atomic state, if any

Parameters

struct drm_atomic_state *state
global atomic state

struct drm_dp_mst_topology_mgr *mgr
MST topology manager, also the private object in this case

Description

This function wraps [drm_atomic_get_old_private_obj_state\(\)](#) passing in the MST atomic state vtable so that the private object state returned is that of a MST topology object.

The old MST topology state, or NULL if there's no topology state for this MST mgr in the global atomic state

Return

struct [drm_dp_mst_topology_state](#) ***drm_atomic_get_new_mst_topology_state**(struct [drm_atomic_state](#) *state, struct [drm_dp_mst_topology_mgr](#) *mgr)

get new MST topology state in atomic state, if any

Parameters

struct drm_atomic_state *state
global atomic state

struct drm_dp_mst_topology_mgr *mgr
MST topology manager, also the private object in this case

Description

This function wraps [drm_atomic_get_new_private_obj_state\(\)](#) passing in the MST atomic state vtable so that the private object state returned is that of a MST topology object.

The new MST topology state, or NULL if there's no topology state for this MST mgr in the global atomic state

Return

int **drm_dp_mst_topology_mgr_init**(struct [drm_dp_mst_topology_mgr](#) *mgr, struct [drm_device](#) *dev, struct [drm_dp_aux](#) *aux, int max_dpcd_transaction_bytes, int max_payloads, int conn_base_id)

initialise a topology manager

Parameters

struct drm_dp_mst_topology_mgr *mgr
manager struct to initialise

struct drm_device *dev
device providing this structure - for i2c addition.

struct drm_dp_aux *aux
DP helper aux channel to talk to this device

int max_dpcd_transaction_bytes

hw specific DPCD transaction limit

int max_payloads

maximum number of payloads this GPU can source

int conn_base_id

the connector object ID the MST device is connected to.

Description

Return 0 for success, or negative error code on failure

void **drm_dp_mst_topology_mgr_destroy**(struct *drm_dp_mst_topology_mgr* *mgr)

destroy topology manager.

Parameters

struct drm_dp_mst_topology_mgr *mgr

manager to destroy

struct *drm_dp_aux* ***drm_dp_mst_dsc_aux_for_port**(struct *drm_dp_mst_port* *port)

Find the correct aux for DSC

Parameters

struct drm_dp_mst_port *port

The port to check. A leaf of the MST tree with an attached display.

Description

Depending on the situation, DSC may be enabled via the endpoint aux, the immediately upstream aux, or the connector's physical aux.

This is both the correct aux to read DSC_CAPABILITY and the correct aux to write DSC_ENABLED.

This operation can be expensive (up to four aux reads), so the caller should cache the return.

Return

NULL if DSC cannot be enabled on this port, otherwise the aux device

5.15.3 Topology Lifetime Internals

These functions aren't exported to drivers, but are documented here to help make the MST topology helpers easier to understand

void **drm_dp_mst_get_mstb_malloc**(struct *drm_dp_mst_branch* *mstb)

Increment the malloc refcount of a branch device

Parameters

struct drm_dp_mst_branch *mstb

The *struct drm_dp_mst_branch* to increment the malloc refcount of

Description

Increments *drm_dp_mst_branch.alloc_kref*. When *drm_dp_mst_branch.alloc_kref* reaches 0, the memory allocation for **mstb** will be released and **mstb** may no longer be used.

See also: `drm_dp_mst_put_mstb_malloc()`

void **drm_dp_mst_put_mstb_malloc**(struct *drm_dp_mst_branch* *mstb)

Decrement the malloc refcount of a branch device

Parameters

struct drm_dp_mst_branch *mstb

The *struct drm_dp_mst_branch* to decrement the malloc refcount of

Description

Decrements *drm_dp_mst_branch.malloc_kref*. When *drm_dp_mst_branch.malloc_kref* reaches 0, the memory allocation for **mstb** will be released and **mstb** may no longer be used.

See also: `drm_dp_mst_get_mstb_malloc()`

int **drm_dp_mst_topology_try_get_mstb**(struct *drm_dp_mst_branch* *mstb)

Increment the topology refcount of a branch device unless it's zero

Parameters

struct drm_dp_mst_branch *mstb

struct drm_dp_mst_branch to increment the topology refcount of

Description

Attempts to grab a topology reference to **mstb**, if it hasn't yet been removed from the topology (e.g. *drm_dp_mst_branch.topology_kref* has reached 0). Holding a topology reference implies that a malloc reference will be held to **mstb** as long as the user holds the topology reference.

Care should be taken to ensure that the user has at least one malloc reference to **mstb**. If you already have a topology reference to **mstb**, you should use `drm_dp_mst_topology_get_mstb()` instead.

See also: `drm_dp_mst_topology_get_mstb()` `drm_dp_mst_topology_put_mstb()`

Return

- 1: A topology reference was grabbed successfully
- 0: **port** is no longer in the topology, no reference was grabbed

void **drm_dp_mst_topology_get_mstb**(struct *drm_dp_mst_branch* *mstb)

Increment the topology refcount of a branch device

Parameters

struct drm_dp_mst_branch *mstb

The *struct drm_dp_mst_branch* to increment the topology refcount of

Description

Increments *drm_dp_mst_branch.topology_refcount* without checking whether or not it's already reached 0. This is only valid to use in scenarios where you are already guaranteed to have at least one active topology reference to **mstb**. Otherwise, `drm_dp_mst_topology_try_get_mstb()` must be used.

See also: `drm_dp_mst_topology_try_get_mstb()` `drm_dp_mst_topology_put_mstb()`

void **drm_dp_mst_topology_put_mstb**(struct *drm_dp_mst_branch* *mstb)
 release a topology reference to a branch device

Parameters

struct drm_dp_mst_branch *mstb
 The *struct drm_dp_mst_branch* to release the topology reference from

Description

Releases a topology reference from **mstb** by decrementing *drm_dp_mst_branch.topology_kref*.

See also: *drm_dp_mst_topology_try_get_mstb()* *drm_dp_mst_topology_get_mstb()*

int **drm_dp_mst_topology_try_get_port**(struct *drm_dp_mst_port* *port)
 Increment the topology refcount of a port unless it's zero

Parameters

struct drm_dp_mst_port *port
struct drm_dp_mst_port to increment the topology refcount of

Description

Attempts to grab a topology reference to **port**, if it hasn't yet been removed from the topology (e.g. *drm_dp_mst_port.topology_kref* has reached 0). Holding a topology reference implies that a malloc reference will be held to **port** as long as the user holds the topology reference.

Care should be taken to ensure that the user has at least one malloc reference to **port**. If you already have a topology reference to **port**, you should use *drm_dp_mst_topology_get_port()* instead.

See also: *drm_dp_mst_topology_get_port()* *drm_dp_mst_topology_put_port()*

Return

- 1: A topology reference was grabbed successfully
- 0: **port** is no longer in the topology, no reference was grabbed

void **drm_dp_mst_topology_get_port**(struct *drm_dp_mst_port* *port)
 Increment the topology refcount of a port

Parameters

struct drm_dp_mst_port *port
 The *struct drm_dp_mst_port* to increment the topology refcount of

Description

Increments *drm_dp_mst_port.topology_refcount* without checking whether or not it's already reached 0. This is only valid to use in scenarios where you are already guaranteed to have at least one active topology reference to **port**. Otherwise, *drm_dp_mst_topology_try_get_port()* must be used.

See also: *drm_dp_mst_topology_try_get_port()* *drm_dp_mst_topology_put_port()*

void **drm_dp_mst_topology_put_port**(struct *drm_dp_mst_port* *port)
 release a topology reference to a port

Parameters

struct drm_dp_mst_port *port

The *struct drm_dp_mst_port* to release the topology reference from

Description

Releases a topology reference from **port** by decrementing *drm_dp_mst_port.topology_kref*.

See also: *drm_dp_mst_topology_try_get_port()* *drm_dp_mst_topology_get_port()*

5.16 MIPI DBI Helper Functions Reference

This library provides helpers for MIPI Display Bus Interface (DBI) compatible display controllers.

Many controllers for tiny lcd displays are MIPI compliant and can use this library. If a controller uses registers 0x2A and 0x2B to set the area to update and uses register 0x2C to write to frame memory, it is most likely MIPI compliant.

Only MIPI Type 1 displays are supported since a full frame memory is needed.

There are 3 MIPI DBI implementation types:

- A. Motorola 6800 type parallel bus
- B. Intel 8080 type parallel bus
- C. SPI type with 3 options:
 - 1. 9-bit with the Data/Command signal as the ninth bit
 - 2. Same as above except it's sent as 16 bits
 - 3. 8-bit with the Data/Command signal as a separate D/CX pin

Currently *mipi_dbi* only supports Type C options 1 and 3 with *mipi_dbi_spi_init()*.

struct mipi_dbi

MIPI DBI interface

Definition:

```
struct mipi_dbi {
    struct mutex cmdlock;
    int (*command)(struct mipi_dbi *dbi, u8 *cmd, u8 *param, size_t num);
    const u8 *read_commands;
    bool swap_bytes;
    struct gpio_desc *reset;
    struct spi_device *spi;
    struct gpio_desc *dc;
    void *tx_buf9;
    size_t tx_buf9_len;
};
```

Members

cmdlock

Command lock

command

Bus specific callback executing commands.

read_commands

Array of read commands terminated by a zero entry.

Reading is disabled if this is NULL.

swap_bytes

Swap bytes in buffer before transfer

reset

Optional reset gpio

spi

SPI device

dc

Optional D/C gpio.

tx_buf9

Buffer used for Option 1 9-bit conversion

tx_buf9_len

Size of tx_buf9.

struct **mipi_dbi_dev**

MIPI DBI device

Definition:

```
struct mipi_dbi_dev {
    struct drm_device drm;
    struct drm_simple_display_pipe pipe;
    struct drm_connector connector;
    struct drm_display_mode mode;
    u16 *tx_buf;
    unsigned int rotation;
    unsigned int left_offset;
    unsigned int top_offset;
    struct backlight_device *backlight;
    struct regulator *regulator;
    struct regulator *io_regulator;
    struct mipi_dbi dbi;
    void *driver_private;
};
```

Members**drm**

DRM device

pipe

Display pipe structure

connector

Connector

mode

Fixed display mode

tx_buf

Buffer used for transfer (copy clip rect area)

rotation

initial rotation in degrees Counter Clock Wise

left_offset

Horizontal offset of the display relative to the controller's driver array

top_offset

Vertical offset of the display relative to the controller's driver array

backlight

backlight device (optional)

regulator

power regulator (Vdd) (optional)

io_regulator

I/O power regulator (Vddi) (optional)

dbi

MIPI DBI interface

driver_private

Driver private data.

Necessary for drivers with private data since `devm_drm_dev_alloc()` can't allocate structures that embed a structure which then again embeds `drm_device`.

mipi_dbi_command

`mipi_dbi_command (dbi, cmd, seq...)`

MIPI DCS command with optional parameter(s)

Parameters**dbi**

MIPI DBI structure

cmd

Command

seq...

Optional parameter(s)

Description

Send MIPI DCS command to the controller. Use `mipi_dbi_command_read()` for get/read.

Return

Zero on success, negative error code on failure.

DRM_MIPI_DBI_SIMPLE_DISPLAY_PIPE_FUNCS

DRM_MIPI_DBI_SIMPLE_DISPLAY_PIPE_FUNCS (enable_)

Initializes *struct drm_simple_display_pipe_funcs* for MIPI-DBI devices

Parameters

enable_

Enable-callback implementation

Description

This macro initializes *struct drm_simple_display_pipe_funcs* with default values for MIPI-DBI-based devices. The only callback that depends on the hardware is **enable**, for which the driver has to provide an implementation. MIPI-based drivers are encouraged to use this macro for initialization.

int **mipi_dbi_command_read**(struct *mipi_dbi* *dbi, u8 cmd, u8 *val)

MIPI DCS read command

Parameters

struct mipi_dbi *dbi

MIPI DBI structure

u8 cmd

Command

u8 *val

Value read

Description

Send MIPI DCS read command to the controller.

Return

Zero on success, negative error code on failure.

int **mipi_dbi_command_buf**(struct *mipi_dbi* *dbi, u8 cmd, u8 *data, size_t len)

MIPI DCS command with parameter(s) in an array

Parameters

struct mipi_dbi *dbi

MIPI DBI structure

u8 cmd

Command

u8 *data

Parameter buffer

size_t len

Buffer length

Return

Zero on success, negative error code on failure.

int **mipi_dbi_buf_copy**(void *dst, struct iosys_map *src, struct *drm_framebuffer* *fb, struct *drm_rect* *clip, bool swap)

Copy a framebuffer, transforming it if necessary

Parameters

void *dst

The destination buffer

struct iosys_map *src

The source buffer

struct drm_framebuffer *fb

The source framebuffer

struct drm_rect *clip

Clipping rectangle of the area to be copied

bool swap

When true, swap MSB/LSB of 16-bit values

Return

Zero on success, negative error code on failure.

enum *drm_mode_status* **mipi_dbi_pipe_mode_valid**(struct *drm_simple_display_pipe* *pipe, const struct *drm_display_mode* *mode)

MIPI DBI mode-valid helper

Parameters

struct drm_simple_display_pipe *pipe

Simple display pipe

const struct drm_display_mode *mode

The mode to test

Description

This function validates a given display mode against the MIPI DBI's hardware display. Drivers can use this as their *drm_simple_display_pipe_funcs->mode_valid* callback.

void **mipi_dbi_pipe_update**(struct *drm_simple_display_pipe* *pipe, struct *drm_plane_state* *old_state)

Display pipe update helper

Parameters

struct drm_simple_display_pipe *pipe

Simple display pipe

struct drm_plane_state *old_state

Old plane state

Description

This function handles framebuffer flushing and vblank events. Drivers can use this as their *drm_simple_display_pipe_funcs->update* callback.

```
void mipi_dbi_enable_flush(struct mipi_dbi_dev *dbidev, struct drm_crtc_state *crtc_state,  
                           struct drm_plane_state *plane_state)
```

MIPI DBI enable helper

Parameters

```
struct mipi_dbi_dev *dbidev
```

MIPI DBI device structure

```
struct drm_crtc_state *crtc_state
```

CRTC state

```
struct drm_plane_state *plane_state
```

Plane state

Description

Flushes the whole framebuffer and enables the backlight. Drivers can use this in their *drm_simple_display_pipe_funcs->enable* callback.

Note

Drivers which don't use *mipi_dbi_pipe_update()* because they have custom framebuffer flushing, can't use this function since they both use the same flushing code.

```
void mipi_dbi_pipe_disable(struct drm_simple_display_pipe *pipe)
```

MIPI DBI pipe disable helper

Parameters

```
struct drm_simple_display_pipe *pipe
```

Display pipe

Description

This function disables backlight if present, if not the display memory is blanked. The regulator is disabled if in use. Drivers can use this as their *drm_simple_display_pipe_funcs->disable* callback.

```
int mipi_dbi_pipe_begin_fb_access(struct drm_simple_display_pipe *pipe, struct  
                                  drm_plane_state *plane_state)
```

MIPI DBI pipe begin-access helper

Parameters

```
struct drm_simple_display_pipe *pipe
```

Display pipe

```
struct drm_plane_state *plane_state
```

Plane state

Description

This function implements struct *drm_simple_display_funcs.begin_fb_access*.

See *drm_gem_begin_shadow_fb_access()* for details and *mipi_dbi_pipe_cleanup_fb()* for cleanup.

Return

0 on success, or a negative errno code otherwise.

```
void mipi_dbi_pipe_end_fb_access(struct drm_simple_display_pipe *pipe, struct  
                                drm_plane_state *plane_state)
```

MIPI DBI pipe end-access helper

Parameters

```
struct drm_simple_display_pipe *pipe  
    Display pipe
```

```
struct drm_plane_state *plane_state  
    Plane state
```

Description

This function implements struct `drm_simple_display_funcs.end_fb_access`.

See [*mipi_dbi_pipe_begin_fb_access\(\)*](#).

```
void mipi_dbi_pipe_reset_plane(struct drm_simple_display_pipe *pipe)  
    MIPI DBI plane-reset helper
```

Parameters

```
struct drm_simple_display_pipe *pipe  
    Display pipe
```

Description

This function implements struct `drm_simple_display_funcs.reset_plane` for MIPI DBI planes.

```
struct drm_plane_state *mipi_dbi_pipe_duplicate_plane_state(struct  
                                                            drm_simple_display_pipe  
                                                            *pipe)
```

duplicates MIPI DBI plane state

Parameters

```
struct drm_simple_display_pipe *pipe  
    Display pipe
```

Description

This function implements struct `drm_simple_display_funcs.duplicate_plane_state` for MIPI DBI planes.

See [*drm_gem_duplicate_shadow_plane_state\(\)*](#) for additional details.

Return

A pointer to a new plane state on success, or NULL otherwise.

```
void mipi_dbi_pipe_destroy_plane_state(struct drm_simple_display_pipe *pipe, struct  
                                         drm_plane_state *plane_state)
```

cleans up MIPI DBI plane state

Parameters

```
struct drm_simple_display_pipe *pipe  
    Display pipe
```



```
struct drm_plane_state *plane_state
```

Plane state

Description

This function implements struct `drm_simple_display_funcs.destroy_plane_state` for MIPI DBI planes.

See [`drm_gem_destroy_shadow_plane_state\(\)`](#) for additional details.

```
int mipi_dbi_dev_init_with_formats(struct mipi_dbi_dev *dbidev, const struct
                                drm_simple_display_pipe_funcs *funcs, const uint32_t
                                *formats, unsigned int format_count, const struct
                                drm_display_mode *mode, unsigned int rotation,
                                size_t tx_buf_size)
```

MIPI DBI device initialization with custom formats

Parameters

```
struct mipi_dbi_dev *dbidev
```

MIPI DBI device structure to initialize

```
const struct drm_simple_display_pipe_funcs *funcs
```

Display pipe functions

```
const uint32_t *formats
```

Array of supported formats (DRM_FORMAT_*).

```
unsigned int format_count
```

Number of elements in **formats**

```
const struct drm_display_mode *mode
```

Display mode

```
unsigned int rotation
```

Initial rotation in degrees Counter Clock Wise

```
size_t tx_buf_size
```

Allocate a transmit buffer of this size.

Description

This function sets up a [`drm_simple_display_pipe`](#) with a [`drm_connector`](#) that has one fixed [`drm_display_mode`](#) which is rotated according to **rotation**. This mode is used to set the mode config min/max width/height properties.

Use [`mipi_dbi_dev_init\(\)`](#) if you don't need custom formats.

Note

Some of the helper functions expects RGB565 to be the default format and the transmit buffer sized to fit that.

Return

Zero on success, negative error code on failure.

```
int mipi_dbi_dev_init(struct mipi_dbi_dev *dbidev, const struct
                    drm_simple_display_pipe_funcs *funcs, const struct
                    drm_display_mode *mode, unsigned int rotation)
```

MIPI DBI device initialization

Parameters

struct mipi_dbi_dev *dbidev

MIPI DBI device structure to initialize

const struct drm_simple_display_pipe_funcs *funcs

Display pipe functions

const struct drm_display_mode *mode

Display mode

unsigned int rotation

Initial rotation in degrees Counter Clock Wise

Description

This function sets up a *drm_simple_display_pipe* with a *drm_connector* that has one fixed *drm_display_mode* which is rotated according to **rotation**. This mode is used to set the mode config min/max width/height properties. Additionally *mipi_dbi.tx_buf* is allocated.

Supported formats: Native RGB565 and emulated XRGB8888.

Return

Zero on success, negative error code on failure.

void **mipi_dbi_hw_reset**(struct *mipi_dbi* *dbi)

Hardware reset of controller

Parameters

struct mipi_dbi *dbi

MIPI DBI structure

Description

Reset controller if the *mipi_dbi->reset* gpio is set.

bool **mipi_dbi_display_is_on**(struct *mipi_dbi* *dbi)

Check if display is on

Parameters

struct mipi_dbi *dbi

MIPI DBI structure

Description

This function checks the Power Mode register (if readable) to see if display output is turned on. This can be used to see if the bootloader has already turned on the display avoiding flicker when the pipeline is enabled.

Return

true if the display can be verified to be on, false otherwise.

int **mipi_dbi_poweron_reset**(struct *mipi_dbi_dev* *dbidev)

MIPI DBI poweron and reset

Parameters

struct mipi_dbi_dev *dbidev

MIPI DBI device structure

Description

This function enables the regulator if used and does a hardware and software reset.

Return

Zero on success, or a negative error code.

int **mipi_dbi_poweron_conditional_reset**(struct *mipi_dbi_dev* *dbidev)
MIPI DBI poweron and conditional reset

Parameters

struct mipi_dbi_dev *dbidev
MIPI DBI device structure

Description

This function enables the regulator if used and if the display is off, it does a hardware and software reset. If *mipi_dbi_display_is_on()* determines that the display is on, no reset is performed.

Return

Zero if the controller was reset, 1 if the display was already on, or a negative error code.

u32 **mipi_dbi_spi_cmd_max_speed**(struct spi_device *spi, size_t len)
get the maximum SPI bus speed

Parameters

struct spi_device *spi
SPI device

size_t len
The transfer buffer length.

Description

Many controllers have a max speed of 10MHz, but can be pushed way beyond that. Increase reliability by running pixel data at max speed and the rest at 10MHz, preventing transfer glitches from messing up the init settings.

int **mipi_dbi_spi_init**(struct spi_device *spi, struct *mipi_dbi* *dbi, struct gpio_desc *dc)
Initialize MIPI DBI SPI interface

Parameters

struct spi_device *spi
SPI device

struct mipi_dbi *dbi
MIPI DBI structure to initialize

struct gpio_desc *dc
D/C gpio (optional)

Description

This function sets *mipi_dbi->command*, enables *mipi_dbi->read_commands* for the usual read commands. It should be followed by a call to *mipi_dbi_dev_init()* or a driver-specific init.

If **dc** is set, a Type C Option 3 interface is assumed, if not Type C Option 1.

If the SPI master driver doesn't support the necessary bits per word, the following transformation is used:

- 9-bit: reorder buffer as 9x 8-bit words, padded with no-op command.
- 16-bit: if big endian send as 8-bit, if little endian swap bytes

Return

Zero on success, negative error code on failure.

```
int mipi_dbi_spi_transfer(struct spi_device *spi, u32 speed_hz, u8 bpw, const void *buf,
                          size_t len)
```

SPI transfer helper

Parameters

struct spi_device *spi

SPI device

u32 speed_hz

Override speed (optional)

u8 bpw

Bits per word

const void *buf

Buffer to transfer

size_t len

Buffer length

Description

This SPI transfer helper breaks up the transfer of **buf** into chunks which the SPI controller driver can handle. The SPI bus must be locked when calling this.

Return

Zero on success, negative error code on failure.

```
void mipi_dbi_debugfs_init(struct drm_minor *minor)
```

Create debugfs entries

Parameters

struct drm_minor *minor

DRM minor

Description

This function creates a 'command' debugfs file for sending commands to the controller or getting the read command values. Drivers can use this as their *drm_driver->debugfs_init* callback.

5.17 MIPI DSI Helper Functions Reference

These functions contain some common logic and helpers to deal with MIPI DSI peripherals.

Helpers are provided for a number of standard MIPI DSI command as well as a subset of the MIPI DCS command set.

struct **mipi_dsi_msg**
 read/write DSI buffer

Definition:

```
struct mipi_dsi_msg {  
    u8 channel;  
    u8 type;  
    u16 flags;  
    size_t tx_len;  
    const void *tx_buf;  
    size_t rx_len;  
    void *rx_buf;  
};
```

Members

channel
 virtual channel id

type
 payload data type

flags
 flags controlling this message transmission

tx_len
 length of **tx_buf**

tx_buf
 data to be written

rx_len
 length of **rx_buf**

rx_buf
 data to be read, or NULL

struct **mipi_dsi_packet**
 represents a MIPI DSI packet in protocol format

Definition:

```
struct mipi_dsi_packet {  
    size_t size;  
    u8 header[4];  
    size_t payload_length;  
    const u8 *payload;  
};
```

Members

size

size (in bytes) of the packet

header

the four bytes that make up the header (Data ID, Word Count or Packet Data, and ECC)

payload_length

number of bytes in the payload

payload

a pointer to a buffer containing the payload, if any

struct **mipi_dsi_host_ops**

DSI bus operations

Definition:

```
struct mipi_dsi_host_ops {
    int (*attach)(struct mipi_dsi_host *host, struct mipi_dsi_device *dsi);
    int (*detach)(struct mipi_dsi_host *host, struct mipi_dsi_device *dsi);
    ssize_t (*transfer)(struct mipi_dsi_host *host, const struct mipi_dsi_msg
→ *msg);
};
```

Members

attach

attach DSI device to DSI host

detach

detach DSI device from DSI host

transfer

transmit a DSI packet

Description

DSI packets transmitted by `.transfer()` are passed in as `mipi_dsi_msg` structures. This structure contains information about the type of packet being transmitted as well as the transmit and receive buffers. When an error is encountered during transmission, this function will return a negative error code. On success it shall return the number of bytes transmitted for write packets or the number of bytes received for read packets.

Note that typically DSI packet transmission is atomic, so the `.transfer()` function will seldomly return anything other than the number of bytes contained in the transmit buffer on success.

Also note that those callbacks can be called no matter the state the host is in. Drivers that need the underlying device to be powered to perform these operations will first need to make sure it's been properly enabled.

struct **mipi_dsi_host**

DSI host device

Definition:

```
struct mipi_dsi_host {
    struct device *dev;
    const struct mipi_dsi_host_ops *ops;
    struct list_head list;
};
```

Members

dev

driver model device node for this DSI host

ops

DSI host operations

list

list management

struct **mipi_dsi_device_info**

template for creating a mipi_dsi_device

Definition:

```
struct mipi_dsi_device_info {
    char type[DSI_DEV_NAME_SIZE];
    u32 channel;
    struct device_node *node;
};
```

Members

type

DSI peripheral chip type

channel

DSI virtual channel assigned to peripheral

node

pointer to OF device node or NULL

Description

This is populated and passed to `mipi_dsi_device_new` to create a new DSI device

struct **mipi_dsi_device**

DSI peripheral device

Definition:

```
struct mipi_dsi_device {
    struct mipi_dsi_host *host;
    struct device dev;
    bool attached;
    char name[DSI_DEV_NAME_SIZE];
    unsigned int channel;
    unsigned int lanes;
    enum mipi_dsi_pixel_format format;
    unsigned long mode_flags;
};
```

```
unsigned long hs_rate;
unsigned long lp_rate;
struct drm_dsc_config *dsc;
};
```

Members

host

DSI host for this peripheral

dev

driver model device node for this peripheral

attached

the DSI device has been successfully attached

name

DSI peripheral chip type

channel

virtual channel assigned to the peripheral

lanes

number of active data lanes

format

pixel format for video mode

mode_flags

DSI operation mode related flags

hs_rate

maximum lane frequency for high speed mode in hertz, this should be set to the real limits of the hardware, zero is only accepted for legacy drivers

lp_rate

maximum lane frequency for low power mode in hertz, this should be set to the real limits of the hardware, zero is only accepted for legacy drivers

dsc

panel/bridge DSC pps payload to be sent

int **mipi_dsi_pixel_format_to_bpp**(enum mipi_dsi_pixel_format fmt)

obtain the number of bits per pixel for any given pixel format defined by the MIPI DSI specification

Parameters

enum **mipi_dsi_pixel_format** fmt

MIPI DSI pixel format

Return

The number of bits per pixel of the given pixel format.

enum **mipi_dsi_dcs_tear_mode**

Tearing Effect Output Line mode

Constants

MIPI_DSI_DCS_TEAR_MODE_VBLANK

the TE output line consists of V-Blanking information only

MIPI_DSI_DCS_TEAR_MODE_VHBLANK

the TE output line consists of both V-Blanking and H-Blanking information

mipi_dsi_generic_write_seq

`mipi_dsi_generic_write_seq (dsi, seq...)`

transmit data using a generic write packet

Parameters**dsi**

DSI peripheral device

seq...

buffer containing the payload

mipi_dsi_dcs_write_seq

`mipi_dsi_dcs_write_seq (dsi, cmd, seq...)`

transmit a DCS command with payload

Parameters**dsi**

DSI peripheral device

cmd

Command

seq...

buffer containing data to be transmitted

struct **mipi_dsi_driver**

DSI driver

Definition:

```
struct mipi_dsi_driver {
    struct device_driver driver;
    int(*probe)(struct mipi_dsi_device *dsi);
    void (*remove)(struct mipi_dsi_device *dsi);
    void (*shutdown)(struct mipi_dsi_device *dsi);
};
```

Members**driver**

device driver model driver

probe

callback for device binding

remove

callback for device unbinding

shutdown

called at shutdown time to quiesce the device

struct *mipi_dsi_device* ***of_find_mipi_dsi_device_by_node**(struct device_node *np)
find the MIPI DSI device matching a device tree node

Parameters

struct device_node *np
device tree node

Return

A pointer to the MIPI DSI device corresponding to np or NULL if no
such device exists (or has not been registered yet).

struct *mipi_dsi_device* ***mipi_dsi_device_register_full**(struct *mipi_dsi_host* *host, const
struct *mipi_dsi_device_info* *info)
create a MIPI DSI device

Parameters

struct mipi_dsi_host *host
DSI host to which this device is connected
const struct mipi_dsi_device_info *info
pointer to template containing DSI device information

Description

Create a MIPI DSI device by using the device information provided by *mipi_dsi_device_info* template

Return

A pointer to the newly created MIPI DSI device, or, a pointer encoded with an error

void **mipi_dsi_device_unregister**(struct *mipi_dsi_device* *dsi)
unregister MIPI DSI device

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device
struct *mipi_dsi_device* ***devm_mipi_dsi_device_register_full**(struct device *dev, struct
mipi_dsi_host *host, const
struct *mipi_dsi_device_info* *info)
create a managed MIPI DSI device

Parameters

struct device *dev
device to tie the MIPI-DSI device lifetime to
struct mipi_dsi_host *host
DSI host to which this device is connected
const struct mipi_dsi_device_info *info
pointer to template containing DSI device information

Description

Create a MIPI DSI device by using the device information provided by `mipi_dsi_device_info` template

This is the managed version of `mipi_dsi_device_register_full()` which automatically calls `mipi_dsi_device_unregister()` when **dev** is unbound.

Return

A pointer to the newly created MIPI DSI device, or, a pointer encoded with an error

struct *mipi_dsi_host* ***of_find_mipi_dsi_host_by_node**(struct device_node *node)
find the MIPI DSI host matching a device tree node

Parameters

struct device_node *node
device tree node

Return

A pointer to the MIPI DSI host corresponding to **node** or NULL if no such device exists (or has not been registered yet).

int **mipi_dsi_attach**(struct *mipi_dsi_device* *dsi)
attach a DSI device to its DSI host

Parameters

struct *mipi_dsi_device* *dsi
DSI peripheral

int **mipi_dsi_detach**(struct *mipi_dsi_device* *dsi)
detach a DSI device from its DSI host

Parameters

struct *mipi_dsi_device* *dsi
DSI peripheral

int **devm_mipi_dsi_attach**(struct device *dev, struct *mipi_dsi_device* *dsi)
Attach a MIPI-DSI device to its DSI Host

Parameters

struct device *dev
device to tie the MIPI-DSI device attachment lifetime to

struct *mipi_dsi_device* *dsi
DSI peripheral

Description

This is the managed version of `mipi_dsi_attach()` which automatically calls `mipi_dsi_detach()` when **dev** is unbound.

Return

0 on success, a negative error code on failure.

bool **mipi_dsi_packet_format_is_short**(u8 type)
check if a packet is of the short format

Parameters

u8 type
MIPI DSI data type of the packet

Return

true if the packet for the given data type is a short packet, false otherwise.

bool **mipi_dsi_packet_format_is_long**(u8 type)
check if a packet is of the long format

Parameters

u8 type
MIPI DSI data type of the packet

Return

true if the packet for the given data type is a long packet, false otherwise.

int **mipi_dsi_create_packet**(struct *mipi_dsi_packet* *packet, const struct *mipi_dsi_msg* *msg)
create a packet from a message according to the DSI protocol

Parameters

struct mipi_dsi_packet *packet
pointer to a DSI packet structure

const struct mipi_dsi_msg *msg
message to translate into a packet

Return

0 on success or a negative error code on failure.

int **mipi_dsi_shutdown_peripheral**(struct *mipi_dsi_device* *dsi)
sends a Shutdown Peripheral command

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

Return

0 on success or a negative error code on failure.

int **mipi_dsi_turn_on_peripheral**(struct *mipi_dsi_device* *dsi)
sends a Turn On Peripheral command

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

Return

0 on success or a negative error code on failure.

`ssize_t mipi_dsi_compression_mode(struct mipi_dsi_device *dsi, bool enable)`
enable/disable DSC on the peripheral

Parameters

`struct mipi_dsi_device *dsi`
DSI peripheral device

`bool enable`
Whether to enable or disable the DSC

Description

Enable or disable Display Stream Compression on the peripheral using the default Picture Parameter Set and VESA DSC 1.1 algorithm.

Return

0 on success or a negative error code on failure.

`ssize_t mipi_dsi_picture_parameter_set(struct mipi_dsi_device *dsi, const struct drm_dsc_picture_parameter_set *pps)`
transmit the DSC PPS to the peripheral

Parameters

`struct mipi_dsi_device *dsi`
DSI peripheral device

`const struct drm_dsc_picture_parameter_set *pps`
VESA DSC 1.1 Picture Parameter Set

Description

Transmit the VESA DSC 1.1 Picture Parameter Set to the peripheral.

Return

0 on success or a negative error code on failure.

`ssize_t mipi_dsi_generic_write(struct mipi_dsi_device *dsi, const void *payload, size_t size)`
transmit data using a generic write packet

Parameters

`struct mipi_dsi_device *dsi`
DSI peripheral device

`const void *payload`
buffer containing the payload

`size_t size`
size of payload buffer

Description

This function will automatically choose the right data type depending on the payload length.

Return

The number of bytes transmitted on success or a negative error code on failure.

`ssize_t mipi_dsi_generic_read`(struct *mipi_dsi_device* *dsi, const void *params, size_t num_params, void *data, size_t size)

receive data using a generic read packet

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

const void *params

buffer containing the request parameters

size_t num_params

number of request parameters

void *data

buffer in which to return the received data

size_t size

size of receive buffer

Description

This function will automatically choose the right data type depending on the number of parameters passed in.

Return

The number of bytes successfully read or a negative error code on failure.

`ssize_t mipi_dsi_dcs_write_buffer`(struct *mipi_dsi_device* *dsi, const void *data, size_t len)

transmit a DCS command with payload

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

const void *data

buffer containing data to be transmitted

size_t len

size of transmission buffer

Description

This function will automatically choose the right data type depending on the command payload length.

Return

The number of bytes successfully transmitted or a negative error code on failure.

`ssize_t mipi_dsi_dcs_write`(struct *mipi_dsi_device* *dsi, u8 cmd, const void *data, size_t len)

send DCS write command

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

u8 cmd

DCS command

const void *data

buffer containing the command payload

size_t len

command payload length

Description

This function will automatically choose the right data type depending on the command payload length.

Return

The number of bytes successfully transmitted or a negative error code on failure.

ssize_t **mipi_dsi_dcs_read**(struct *mipi_dsi_device* *dsi, u8 cmd, void *data, size_t len)

send DCS read request command

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

u8 cmd

DCS command

void *data

buffer in which to receive data

size_t len

size of receive buffer

Return

The number of bytes read or a negative error code on failure.

int **mipi_dsi_dcs_nop**(struct *mipi_dsi_device* *dsi)

send DCS nop packet

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_soft_reset**(struct *mipi_dsi_device* *dsi)

perform a software reset of the display module

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_get_power_mode**(struct *mipi_dsi_device* *dsi, u8 *mode)
query the display module's current power mode

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

u8 *mode
return location for the current power mode

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_get_pixel_format**(struct *mipi_dsi_device* *dsi, u8 *format)
gets the pixel format for the RGB image data used by the interface

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

u8 *format
return location for the pixel format

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_enter_sleep_mode**(struct *mipi_dsi_device* *dsi)
disable all unnecessary blocks inside the display module except interface communication

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_exit_sleep_mode**(struct *mipi_dsi_device* *dsi)
enable all blocks inside the display module

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_set_display_off**(struct *mipi_dsi_device* *dsi)
stop displaying the image data on the display device

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_set_display_on**(struct *mipi_dsi_device* *dsi)
start displaying the image data on the display device

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

Return

0 on success or a negative error code on failure

int **mipi_dsi_dcs_set_column_address**(struct *mipi_dsi_device* *dsi, u16 start, u16 end)
define the column extent of the frame memory accessed by the host processor

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

u16 start
first column of frame memory

u16 end
last column of frame memory

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_set_page_address**(struct *mipi_dsi_device* *dsi, u16 start, u16 end)
define the page extent of the frame memory accessed by the host processor

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

u16 start
first page of frame memory

u16 end
last page of frame memory

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_set_tear_off**(struct *mipi_dsi_device* *dsi)
turn off the display module's Tearing Effect output signal on the TE signal line

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

Return

0 on success or a negative error code on failure

int **mipi_dsi_dcs_set_tear_on**(struct *mipi_dsi_device* *dsi, enum *mipi_dsi_dcs_tear_mode* mode)

turn on the display module's Tearing Effect output signal on the TE signal line.

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

enum mipi_dsi_dcs_tear_mode mode

the Tearing Effect Output Line mode

Return

0 on success or a negative error code on failure

int **mipi_dsi_dcs_set_pixel_format**(struct *mipi_dsi_device* *dsi, u8 format)

sets the pixel format for the RGB image data used by the interface

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

u8 format

pixel format

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_set_tear_scanline**(struct *mipi_dsi_device* *dsi, u16 scanline)

set the scanline to use as trigger for the Tearing Effect output signal of the display module

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

u16 scanline

scanline to use as trigger

Return

0 on success or a negative error code on failure

int **mipi_dsi_dcs_set_display_brightness**(struct *mipi_dsi_device* *dsi, u16 brightness)

sets the brightness value of the display

Parameters

struct mipi_dsi_device *dsi

DSI peripheral device

u16 brightness

brightness value

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_get_display_brightness**(struct *mipi_dsi_device* *dsi, u16 *brightness)
gets the current brightness value of the display

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

u16 *brightness
brightness value

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_set_display_brightness_large**(struct *mipi_dsi_device* *dsi, u16
brightness)
sets the 16-bit brightness value of the display

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

u16 brightness
brightness value

Return

0 on success or a negative error code on failure.

int **mipi_dsi_dcs_get_display_brightness_large**(struct *mipi_dsi_device* *dsi, u16
*brightness)
gets the current 16-bit brightness value of the display

Parameters

struct mipi_dsi_device *dsi
DSI peripheral device

u16 *brightness
brightness value

Return

0 on success or a negative error code on failure.

int **mipi_dsi_driver_register_full**(struct *mipi_dsi_driver* *drv, struct module *owner)
register a driver for DSI devices

Parameters

struct mipi_dsi_driver *drv
DSI driver structure

struct module *owner
owner module

Return

0 on success or a negative error code on failure.

void **mipi_dsi_driver_unregister**(struct *mipi_dsi_driver* *drv)
unregister a driver for DSI devices

Parameters

struct mipi_dsi_driver *drv
DSI driver structure

Return

0 on success or a negative error code on failure.

5.18 Display Stream Compression Helper Functions Reference

VESA specification for DP 1.4 adds a new feature called Display Stream Compression (DSC) used to compress the pixel bits before sending it on DP/eDP/MIPI DSI interface. DSC is required to be enabled so that the existing display interfaces can support high resolutions at higher frames rates using the maximum available link capacity of these interfaces.

These functions contain some common logic and helpers to deal with VESA Display Stream Compression standard required for DSC on Display Port/eDP or MIPI display interfaces.

struct **drm_dsc_rc_range_parameters**
DSC Rate Control range parameters

Definition:

```
struct drm_dsc_rc_range_parameters {  
    u8 range_min_qp;  
    u8 range_max_qp;  
    u8 range_bpg_offset;  
};
```

Members

range_min_qp
Min Quantization Parameters allowed for this range

range_max_qp
Max Quantization Parameters allowed for this range

range_bpg_offset
Bits/group offset to apply to target for this group

Description

This defines different rate control parameters used by the DSC engine to compress the frame.

struct **drm_dsc_config**
Parameters required to configure DSC

Definition:

```
struct drm_dsc_config {  
    u8 line_buf_depth;  
    u8 bits_per_component;
```

```

bool convert_rgb;
u8 slice_count;
u16 slice_width;
u16 slice_height;
bool simple_422;
u16 pic_width;
u16 pic_height;
u8 rc_tgt_offset_high;
u8 rc_tgt_offset_low;
u16 bits_per_pixel;
u8 rc_edge_factor;
u8 rc_quant_incr_limit1;
u8 rc_quant_incr_limit0;
u16 initial_xmit_delay;
u16 initial_dec_delay;
bool block_pred_enable;
u8 first_line_bpg_offset;
u16 initial_offset;
u16 rc_buf_thresh[DSC_NUM_BUF_RANGES - 1];
struct drm_dsc_rc_range_parameters rc_range_params[DSC_NUM_BUF_RANGES];
u16 rc_model_size;
u8 flatness_min_qp;
u8 flatness_max_qp;
u8 initial_scale_value;
u16 scale_decrement_interval;
u16 scale_increment_interval;
u16 nfl_bpg_offset;
u16 slice_bpg_offset;
u16 final_offset;
bool vbr_enable;
u8 mux_word_size;
u16 slice_chunk_size;
u16 rc_bits;
u8 dsc_version_minor;
u8 dsc_version_major;
bool native_422;
bool native_420;
u8 second_line_bpg_offset;
u16 nsl_bpg_offset;
u16 second_line_offset_adj;
};

```

Members

line_buf_depth

Bits per component for previous reconstructed line buffer

bits_per_component

Bits per component to code (8/10/12)

convert_rgb

Flag to indicate if RGB - YCoCg conversion is needed True if RGB input, False if YCoCg

input

slice_count

Number fo slices per line used by the DSC encoder

slice_width

Width of each slice in pixels

slice_height

Slice height in pixels

simple_422

True if simple 4_2_2 mode is enabled else False

pic_width

Width of the input display frame in pixels

pic_height

Vertical height of the input display frame

rc_tgt_offset_high

Offset to bits/group used by RC to determine QP adjustment

rc_tgt_offset_low

Offset to bits/group used by RC to determine QP adjustment

bits_per_pixel

Target bits per pixel with 4 fractional bits, $\text{bits_per_pixel} \ll 4$

rc_edge_factor

Factor to determine if an edge is present based on the bits produced

rc_quant_incr_limit1

Slow down incrementing once the range reaches this value

rc_quant_incr_limit0

Slow down incrementing once the range reaches this value

initial_xmit_delay

Number of pixels to delay the initial transmission

initial_dec_delay

Initial decoder delay, number of pixel times that the decoder accumulates data in its rate buffer before starting to decode and output pixels.

block_pred_enable

True if block prediction is used to code any groups within the picture. False if BP not used

first_line_bpg_offset

Number of additional bits allocated for each group on the first line of slice.

initial_offset

Value to use for RC model offset at slice start

rc_buf_thresh

Thresholds defining each of the buffer ranges

rc_range_params

Parameters for each of the RC ranges defined in *struct drm_dsc_rc_range_parameters*

rc_model_size

Total size of RC model

flatness_min_qp

Minimum QP where flatness information is sent

flatness_max_qp

Maximum QP where flatness information is sent

initial_scale_value

Initial value for the scale factor

scale_decrement_interval

Specifies number of group times between decrementing the scale factor at beginning of a slice.

scale_increment_interval

Number of group times between incrementing the scale factor value used at the beginning of a slice.

nfl_bpg_offset

Non first line BPG offset to be used

slice_bpg_offset

BPG offset used to enforce slice bit

final_offset

Final RC linear transformation offset value

vbr_enable

True if VBR mode is enabled, false if disabled

mux_word_size

Mux word size (in bits) for SSM mode

slice_chunk_size

The (max) size in bytes of the “chunks” that are used in slice multiplexing.

rc_bits

Rate control buffer size in bits

dsc_version_minor

DSC minor version

dsc_version_major

DSC major version

native_422

True if Native 4:2:2 supported, else false

native_420

True if Native 4:2:0 supported else false.

second_line_bpg_offset

Additional bits/grp for second line of slice for native 4:2:0

nsl_bpg_offset

Num of bits deallocated for each grp that is not in second line of slice

second_line_offset_adj

Offset adjustment for second line in Native 4:2:0 mode

Description

Driver populates this structure with all the parameters required to configure the display stream compression on the source.

struct **drm_dsc_picture_parameter_set**

Represents 128 bytes of Picture Parameter Set

Definition:

```
struct drm_dsc_picture_parameter_set {
    u8 dsc_version;
    u8 pps_identifier;
    u8 pps_reserved;
    u8 pps_3;
    u8 pps_4;
    u8 bits_per_pixel_low;
    __be16 pic_height;
    __be16 pic_width;
    __be16 slice_height;
    __be16 slice_width;
    __be16 chunk_size;
    u8 initial_xmit_delay_high;
    u8 initial_xmit_delay_low;
    __be16 initial_dec_delay;
    u8 pps20_reserved;
    u8 initial_scale_value;
    __be16 scale_increment_interval;
    u8 scale_decrement_interval_high;
    u8 scale_decrement_interval_low;
    u8 pps26_reserved;
    u8 first_line_bpg_offset;
    __be16 nfl_bpg_offset;
    __be16 slice_bpg_offset;
    __be16 initial_offset;
    __be16 final_offset;
    u8 flatness_min_qp;
    u8 flatness_max_qp;
    __be16 rc_model_size;
    u8 rc_edge_factor;
    u8 rc_quant_incr_limit0;
    u8 rc_quant_incr_limit1;
    u8 rc_tgt_offset;
    u8 rc_buf_thresh[DSC_NUM_BUF_RANGES - 1];
    __be16 rc_range_parameters[DSC_NUM_BUF_RANGES];
    u8 native_422_420;
    u8 second_line_bpg_offset;
    __be16 nsl_bpg_offset;
    __be16 second_line_offset_adj;
    u32 pps_long_94_reserved;
    u32 pps_long_98_reserved;
    u32 pps_long_102_reserved;
    u32 pps_long_106_reserved;
```



```

u32 pps_long_110_reserved;
u32 pps_long_114_reserved;
u32 pps_long_118_reserved;
u32 pps_long_122_reserved;
__be16 pps_short_126_reserved;
};

```

Members

dsc_version

PPS0[3:0] - dsc_version_minor: Contains Minor version of DSC PPS0[7:4] - dsc_version_major: Contains major version of DSC

pps_identifier

PPS1[7:0] - Application specific identifier that can be used to differentiate between different PPS tables.

pps_reserved

PPS2[7:0]- RESERVED Byte

pps_3

PPS3[3:0] - linebuf_depth: Contains linebuffer bit depth used to generate the bitstream. (0x0 - 16 bits for DSC 1.2, 0x8 - 8 bits, 0xA - 10 bits, 0xB - 11 bits, 0xC - 12 bits, 0xD - 13 bits, 0xE - 14 bits for DSC1.2, 0xF - 14 bits for DSC 1.2. PPS3[7:4] - bits_per_component: Bits per component for the original pixels of the encoded picture. 0x0 = 16bpc (allowed only when dsc_version_minor = 0x2) 0x8 = 8bpc, 0xA = 10bpc, 0xC = 12bpc, 0xE = 14bpc (also allowed only when dsc_minor_version = 0x2)

pps_4

PPS4[1:0] -These are the most significant 2 bits of compressed BPP bits_per_pixel[9:0] syntax element. PPS4[2] - vbr_enable: 0 = VBR disabled, 1 = VBR enabled PPS4[3] - simple_422: Indicates if decoder drops samples to reconstruct the 4:2:2 picture. PPS4[4] - Convert_rgb: Indicates if DSC color space conversion is active. PPS4[5] - blobk_pred_enable: Indicates if BP is used to code any groups in picture PPS4[7:6] - Reserved bits

bits_per_pixel_low

PPS5[7:0] - This indicates the lower significant 8 bits of the compressed BPP bits_per_pixel[9:0] element.

pic_height

PPS6[7:0], PPS7[7:0] -pic_height: Specifies the number of pixel rows within the raster.

pic_width

PPS8[7:0], PPS9[7:0] - pic_width: Number of pixel columns within the raster.

slice_height

PPS10[7:0], PPS11[7:0] - Slice height in units of pixels.

slice_width

PPS12[7:0], PPS13[7:0] - Slice width in terms of pixels.

chunk_size

PPS14[7:0], PPS15[7:0] - Size in units of bytes of the chunks that are used for slice multiplexing.

initial_xmit_delay_high

PPS16[1:0] - Most Significant two bits of initial transmission delay. It specifies the number of pixel times that the encoder waits before transmitting data from its rate buffer.
PPS16[7:2] - Reserved

initial_xmit_delay_low

PPS17[7:0] - Least significant 8 bits of initial transmission delay.

initial_dec_delay

PPS18[7:0], PPS19[7:0] - Initial decoding delay which is the number of pixel times that the decoder accumulates data in its rate buffer before starting to decode and output pixels.

pps20_reserved

PPS20[7:0] - Reserved

initial_scale_value

PPS21[5:0] - Initial rcXformScale factor used at beginning of a slice. PPS21[7:6] - Reserved

scale_increment_interval

PPS22[7:0], PPS23[7:0] - Number of group times between incrementing the rcXformScale factor at end of a slice.

scale_decrement_interval_high

PPS24[3:0] - Higher 4 bits indicating number of group times between decrementing the rcXformScale factor at beginning of a slice. PPS24[7:4] - Reserved

scale_decrement_interval_low

PPS25[7:0] - Lower 8 bits of scale decrement interval

pps26_reserved

PPS26[7:0]

first_line_bpg_offset

PPS27[4:0] - Number of additional bits that are allocated for each group on first line of a slice. PPS27[7:5] - Reserved

nfl_bpg_offset

PPS28[7:0], PPS29[7:0] - Number of bits including frac bits deallocated for each group for groups after the first line of slice.

slice_bpg_offset

PPS30, PPS31[7:0] - Number of bits that are deallocated for each group to enforce the slice constraint.

initial_offset

PPS32,33[7:0] - Initial value for rcXformOffset

final_offset

PPS34,35[7:0] - Maximum end-of-slice value for rcXformOffset

flatness_min_qp

PPS36[4:0] - Minimum QP at which flatness is signaled and flatness QP adjustment is made.
PPS36[7:5] - Reserved

flatness_max_qp

PPS37[4:0] - Max QP at which flatness is signalled and the flatness adjustment is made.
PPS37[7:5] - Reserved

rc_model_size

PPS38,39[7:0] - Number of bits within RC Model.

rc_edge_factor

PPS40[3:0] - Ratio of current activity vs, previous activity to determine presence of edge.
PPS40[7:4] - Reserved

rc_quant_incr_limit0

PPS41[4:0] - QP threshold used in short term RC PPS41[7:5] - Reserved

rc_quant_incr_limit1

PPS42[4:0] - QP threshold used in short term RC PPS42[7:5] - Reserved

rc_tgt_offset

PPS43[3:0] - Lower end of the variability range around the target bits per group that is allowed by short term RC. PPS43[7:4]- Upper end of the variability range around the target bits per group that is allowed by short term rc.

rc_buf_thresh

PPS44[7:0] - PPS57[7:0] - Specifies the thresholds in RC model for the 15 ranges defined by 14 thresholds.

rc_range_parameters

PPS58[7:0] - PPS87[7:0] Parameters that correspond to each of the 15 ranges.

native_422_420

PPS88[0] - 0 = Native 4:2:2 not used 1 = Native 4:2:2 used PPS88[1] - 0 = Native 4:2:0 not use 1 = Native 4:2:0 used PPS88[7:2] - Reserved 6 bits

second_line_bpg_offset

PPS89[4:0] - Additional bits/group budget for the second line of a slice in Native 4:2:0 mode. Set to 0 if DSC minor version is 1 or native420 is 0. PPS89[7:5] - Reserved

nsl_bpg_offset

PPS90[7:0], PPS91[7:0] - Number of bits that are deallocated for each group that is not in the second line of a slice.

second_line_offset_adj

PPS92[7:0], PPS93[7:0] - Used as offset adjustment for the second line in Native 4:2:0 mode.

pps_long_94_reserved

PPS 94, 95, 96, 97 - Reserved

pps_long_98_reserved

PPS 98, 99, 100, 101 - Reserved

pps_long_102_reserved

PPS 102, 103, 104, 105 - Reserved

pps_long_106_reserved

PPS 106, 107, 108, 109 - reserved

pps_long_110_reserved

PPS 110, 111, 112, 113 - reserved

pps_long_114_reserved

PPS 114 - 117 - reserved

pps_long_118_reserved

PPS 118 - 121 - reserved

pps_long_122_reserved

PPS 122- 125 - reserved

pps_short_126_reserved

PPS 126, 127 - reserved

Description

The VESA DSC standard defines picture parameter set (PPS) which display stream compression encoders must communicate to decoders. The PPS is encapsulated in 128 bytes (PPS 0 through PPS 127). The fields in this structure are as per Table 4.1 in Vesa DSC specification v1.1/v1.2. The PPS fields that span over more than a byte should be stored in Big Endian format.

struct drm_dsc_pps_infotrame

DSC infotrame carrying the Picture Parameter Set Metadata

Definition:

```
struct drm_dsc_pps_infotrame {
    struct dp_sdp_header pps_header;
    struct drm_dsc_picture_parameter_set pps_payload;
};
```

Members**pps_header**Header for PPS as per DP SDP header format of type *struct dp_sdp_header***pps_payload**PPS payload fields as per DSC specification Table 4-1 as represented in *struct drm_dsc_picture_parameter_set***Description**

This structure represents the DSC PPS infotrame required to send the Picture Parameter Set metadata required before enabling VESA Display Stream Compression. This is based on the DP Secondary Data Packet structure and comprises of SDP Header as defined *struct dp_sdp_header* in *drm_dp_helper.h* and PPS payload defined in *struct drm_dsc_picture_parameter_set*.

void drm_dsc_dp_pps_header_init(struct *dp_sdp_header* *pps_header)

Initializes the PPS Header for DisplayPort as per the DP 1.4 spec.

Parameters**struct dp_sdp_header *pps_header**Secondary data packet header for DSC Picture Parameter Set as defined in *struct dp_sdp_header***Description**

DP 1.4 spec defines the secondary data packet for sending the picture parameter infotrames from the source to the sink. This function populates the SDP header defined in *struct dp_sdp_header*.

int **drm_dsc_dp_rc_buffer_size**(u8 rc_buffer_block_size, u8 rc_buffer_size)
 get rc buffer size in bytes

Parameters

u8 rc_buffer_block_size
 block size code, according to DPCD offset 62h

u8 rc_buffer_size
 number of blocks - 1, according to DPCD offset 63h

Return

buffer size in bytes, or 0 on invalid input

void **drm_dsc_pps_payload_pack**(struct *drm_dsc_picture_parameter_set* *pps_payload, const struct *drm_dsc_config* *dsc_cfg)

Populates the DSC PPS

Parameters

struct *drm_dsc_picture_parameter_set* *pps_payload
 Bitwise struct for DSC Picture Parameter Set. This is defined by *struct drm_dsc_picture_parameter_set*

const struct *drm_dsc_config* *dsc_cfg
 DSC Configuration data filled by driver as defined by *struct drm_dsc_config*

Description

DSC source device sends a picture parameter set (PPS) containing the information required by the sink to decode the compressed frame. Driver populates the DSC PPS struct using the DSC configuration parameters in the order expected by the DSC Display Sink device. For the DSC, the sink device expects the PPS payload in big endian format for fields that span more than 1 byte.

void **drm_dsc_set_const_params**(struct *drm_dsc_config* *vdsc_cfg)
 Set DSC parameters considered typically constant across operation modes

Parameters

struct *drm_dsc_config* *vdsc_cfg
 DSC Configuration data partially filled by driver

void **drm_dsc_set_rc_buf_thresh**(struct *drm_dsc_config* *vdsc_cfg)
 Set thresholds for the RC model in accordance with the DSC 1.2 specification.

Parameters

struct *drm_dsc_config* *vdsc_cfg
 DSC Configuration data partially filled by driver

int **drm_dsc_setup_rc_params**(struct *drm_dsc_config* *vdsc_cfg, enum *drm_dsc_params_type* type)
 Set parameters and limits for RC model in accordance with the DSC 1.1 or 1.2 specification and DSC C Model Required bits_per_pixel and bits_per_component to be set before calling this function.

Parameters

struct drm_dsc_config *vdsc_cfg
DSC Configuration data partially filled by driver

enum drm_dsc_params_type type
operating mode and standard to follow

Return

0 or -error code in case of an error

int **drm_dsc_compute_rc_parameters**(struct *drm_dsc_config* *vdsc_cfg)
Write rate control parameters to the dsc configuration defined in *struct drm_dsc_config* in accordance with the DSC 1.2 specification. Some configuration fields must be present beforehand.

Parameters

struct drm_dsc_config *vdsc_cfg
DSC Configuration data partially filled by driver

u32 **drm_dsc_get_bpp_int**(const struct *drm_dsc_config* *vdsc_cfg)
Get integer bits per pixel value for the given DRM DSC config

Parameters

const struct drm_dsc_config *vdsc_cfg
Pointer to DRM DSC config struct

Return

Integer BPP value

u8 **drm_dsc_initial_scale_value**(const struct *drm_dsc_config* *dsc)
Calculate the initial scale value for the given DSC config

Parameters

const struct drm_dsc_config *dsc
Pointer to DRM DSC config struct

Return

Calculated initial scale value

u32 **drm_dsc_flatness_det_thresh**(const struct *drm_dsc_config* *dsc)
Calculate the flatness_det_thresh for the given DSC config

Parameters

const struct drm_dsc_config *dsc
Pointer to DRM DSC config struct

Return

Calculated flatness det thresh value

5.19 Output Probing Helper Functions Reference

This library provides some helper code for output probing. It provides an implementation of the core `drm_connector_funcs.fill_modes` interface with `drm_helper_probe_single_connector_modes()`.

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

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

The probe helpers share the function table structures with other display helper libraries. See `struct drm_connector_helper_funcs` for the details.

void `drm_kms_helper_poll_enable`(struct `drm_device` *dev)
re-enable output polling.

Parameters

struct `drm_device` *dev
drm_device

Description

This function re-enables the output polling work, after it has been temporarily disabled using `drm_kms_helper_poll_disable()`, for example over suspend/resume.

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

If device polling was never initialized before, this call will trigger a warning and return.

Note that calls to enable and disable polling must be strictly ordered, which is automatically the case when they're only call from suspend/resume callbacks.

void `drm_kms_helper_poll_reschedule`(struct `drm_device` *dev)
reschedule the output polling work

Parameters

struct `drm_device` *dev
drm_device

Description

This function reschedules the output polling work, after polling for a connector has been enabled.

Drivers must call this helper after enabling polling for a connector by setting `DRM_CONNECTOR_POLL_CONNECT` / `DRM_CONNECTOR_POLL_DISCONNECT` flags in `drm_connector::polled`. Note that after disabling polling by clearing these flags for a connector will stop the output polling work automatically if the polling is disabled for all other connectors as well.

The function can be called only after polling has been enabled by calling `drm_kms_helper_poll_init()` / `drm_kms_helper_poll_enable()`.

int **drm_helper_probe_detect**(struct *drm_connector* *connector, struct *drm_modeset_acquire_ctx* *ctx, bool force)
probe connector status

Parameters

struct drm_connector *connector
connector to probe

struct drm_modeset_acquire_ctx *ctx
acquire_ctx, or NULL to let this function handle locking.

bool force
Whether destructive probe operations should be performed.

Description

This function calls the detect callbacks of the connector. This function returns *drm_connector_status*, or if **ctx** is set, it might also return -EDEADLK.

int **drm_helper_probe_single_connector_modes**(struct *drm_connector* *connector, uint32_t maxX, uint32_t maxY)
get complete set of display modes

Parameters

struct drm_connector *connector
connector to probe

uint32_t maxX
max width for modes

uint32_t maxY
max height for modes

Description

Based on the helper callbacks implemented by **connector** in struct *drm_connector_helper_funcs* 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 used as a generic implementation of the *drm_connector_funcs.fill_modes()* vfunc for drivers that use the CRTC helpers for output mode filtering and detection.

The basic procedure is as follows

1. All modes currently on the connector's modes list are marked as stale
2. New modes are added to the connector's probed_modes list with *drm_mode_probed_add()*. New modes start their life with status as OK. Modes are added from a single source using the following priority order.
 - *drm_connector_helper_funcs.get_modes* vfunc
 - if the connector status is connector_status_connected, standard VESA DMT modes up to 1024x768 are automatically added (*drm_add_modes_noedid()*)

Finally modes specified via the kernel command line (video=...) are added in addition to what the earlier probes produced (*drm_helper_probe_add_cmdline_mode()*). These modes are generated using the VESA GTF/CVT formulas.

3. Modes are moved from the `probed_modes` list to the modes list. Potential duplicates are merged together (see `drm_connector_list_update()`). After this step the `probed_modes` list will be empty again.
4. Any non-stale mode on the modes list then undergoes validation
 - `drm_mode_validate_basic()` performs basic sanity checks
 - `drm_mode_validate_size()` filters out modes larger than **maxX** and **maxY** (if specified)
 - `drm_mode_validate_flag()` checks the modes against basic connector capabilities (`interlace_allowed`, `doublescan_allowed`, `stereo_allowed`)
 - the optional `drm_connector_helper_funcs.mode_valid` or `drm_connector_helper_funcs.mode_valid_ctx` helpers can perform driver and/or sink specific checks
 - the optional `drm_crtc_helper_funcs.mode_valid`, `drm_bridge_funcs.mode_valid` and `drm_encoder_helper_funcs.mode_valid` helpers can perform driver and/or source specific checks which are also enforced by the modeset/atomic helpers
5. Any mode whose status is not OK is pruned from the connector's modes list, accompanied by a debug message indicating the reason for the mode's rejection (see `drm_mode_prune_invalid()`).

Return

The number of modes found on **connector**.

void **drm_kms_helper_hotplug_event**(struct *drm_device* *dev)
fire off KMS hotplug events

Parameters

struct drm_device *dev
drm_device whose connector state changed

Description

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

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

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

If only a single connector has changed, consider calling `drm_kms_helper_connector_hotplug_event()` instead.

void **drm_kms_helper_connector_hotplug_event**(struct *drm_connector* *connector)
fire off a KMS connector hotplug event

Parameters

struct drm_connector *connector
drm_connector which has changed

Description

This is the same as `drm_kms_helper_hotplug_event()`, except it fires a more fine-grained uevent for a single connector.

bool `drm_kms_helper_is_poll_worker(void)`
is current task an output poll worker?

Parameters

void
no arguments

Description

Determine if current task is an output poll worker. This can be used to select distinct code paths for output polling versus other contexts.

One use case is to avoid a deadlock between the output poll worker and the autosuspend worker wherein the latter waits for polling to finish upon calling `drm_kms_helper_poll_disable()`, while the former waits for runtime suspend to finish upon calling `pm_runtime_get_sync()` in a connector->detect hook.

void `drm_kms_helper_poll_disable(struct drm_device *dev)`
disable output polling

Parameters

struct *drm_device* *dev
drm_device

Description

This function disables the output polling work.

Drivers can call this helper from their device suspend implementation. It is not an error to call this even when output polling isn't enabled or already disabled. Polling is re-enabled by calling `drm_kms_helper_poll_enable()`.

If however, the polling was never initialized, this call will trigger a warning and return

Note that calls to enable and disable polling must be strictly ordered, which is automatically the case when they're only call from suspend/resume callbacks.

void `drm_kms_helper_poll_init(struct drm_device *dev)`
initialize and enable output polling

Parameters

struct *drm_device* *dev
drm_device

Description

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

Drivers can control which connectors are polled by setting the `DRM_CONNECTOR_POLL_CONNECT` and `DRM_CONNECTOR_POLL_DISCONNECT` flags. On connectors where probing live outputs can result in visual distortion drivers should not set

the `DRM_CONNECTOR_POLL_DISCONNECT` flag to avoid this. Connectors which have no flag or only `DRM_CONNECTOR_POLL_HPD` set are completely ignored by the polling logic.

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

void **drm_kms_helper_poll_fini**(struct *drm_device* *dev)
 disable output polling and clean it up

Parameters

struct drm_device *dev
 drm_device

bool **drm_connector_helper_hpd_irq_event**(struct *drm_connector* *connector)
 hotplug processing

Parameters

struct drm_connector *connector
 drm_connector

Description

Drivers can use this helper function to run a detect cycle on a connector which has the `DRM_CONNECTOR_POLL_HPD` flag set in its `polled` member.

This helper function is useful for drivers which can track hotplug interrupts for a single connector. Drivers that want to send a hotplug event for all connectors or can't track hotplug interrupts per connector need to use *drm_helper_hpd_irq_event()*.

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

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

Return

A boolean indicating whether the connector status changed or not

bool **drm_helper_hpd_irq_event**(struct *drm_device* *dev)
 hotplug processing

Parameters

struct drm_device *dev
 drm_device

Description

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

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

Drivers which support hotplug interrupts for each connector individually and which have a more fine-grained detect logic can use *drm_connector_helper_hpd_irq_event()*. Alternatively, they should bypass this code and directly call *drm_kms_helper_hotplug_event()* in case the connector state changed.

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

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

Return

A boolean indicating whether the connector status changed or not

```
enum drm_mode_status drm_crtc_helper_mode_valid_fixed(struct drm_crtc *crtc, const  
                                                    struct drm_display_mode  
                                                    *mode, const struct  
                                                    drm_display_mode  
                                                    *fixed_mode)
```

Validates a display mode

Parameters

struct *drm_crtc* *crtc
the crtc

const struct *drm_display_mode* *mode
the mode to validate

const struct *drm_display_mode* *fixed_mode
the display hardware's mode

Return

MODE_OK on success, or another mode-status code otherwise.

```
int drm_connector_helper_get_modes_from_ddc(struct drm_connector *connector)  
    Updates the connector's EDID property from the connector's DDC channel
```

Parameters

struct *drm_connector* *connector
The connector

Return

The number of detected display modes.

Description

Uses a connector's DDC channel to retrieve EDID data and update the connector's EDID property and display modes. Drivers can use this function to implement struct *drm_connector_helper_funcs.get_modes* for connectors with a DDC channel.

```
int drm_connector_helper_get_modes_fixed(struct drm_connector *connector, const struct  
                                         drm_display_mode *fixed_mode)
```

Duplicates a display mode for a connector

Parameters

struct *drm_connector* *connector
the connector

const struct *drm_display_mode* *fixed_mode
the display hardware's mode

Description

This function duplicates a display modes for a connector. Drivers for hardware that only supports a single fixed mode can use this function in their connector's `get_modes` helper.

Return

The number of created modes.

int **drm_connector_helper_get_modes**(struct *drm_connector* *connector)
Read EDID and update connector.

Parameters

struct drm_connector *connector
The connector

Description

Read the EDID using *drm_edid_read()* (which requires that `connector->ddc` is set), and update the connector using the EDID.

This can be used as the “default” connector helper `.get_modes()` hook if the driver does not need any special processing. This sets the example what custom `.get_modes()` hooks should do regarding EDID read and connector update.

Return

Number of modes.

int **drm_connector_helper_tv_get_modes**(struct *drm_connector* *connector)
Fills the modes available to a TV connector

Parameters

struct drm_connector *connector
The connector

Description

Fills the available modes for a TV connector based on the supported TV modes, and the default mode expressed by the kernel command line.

This can be used as the default TV connector helper `.get_modes()` hook if the driver does not need any special processing.

Return

The number of modes added to the connector.

5.20 EDID Helper Functions Reference

int **drm_eld_mnl**(const uint8_t *eld)
Get ELD monitor name length in bytes.

Parameters

const uint8_t *eld
pointer to an eld memory structure with `mnl` set

const uint8_t ***drm_eld_sad**(const uint8_t *eld)

Get ELD SAD structures.

Parameters

const uint8_t *eld

pointer to an eld memory structure with sad_count set

int **drm_eld_sad_count**(const uint8_t *eld)

Get ELD SAD count.

Parameters

const uint8_t *eld

pointer to an eld memory structure with sad_count set

int **drm_eld_calc_baseline_block_size**(const uint8_t *eld)

Calculate baseline block size in bytes

Parameters

const uint8_t *eld

pointer to an eld memory structure with mnl and sad_count set

Description

This is a helper for determining the payload size of the baseline block, in bytes, for e.g. setting the Baseline_ELD_Len field in the ELD header block.

int **drm_eld_size**(const uint8_t *eld)

Get ELD size in bytes

Parameters

const uint8_t *eld

pointer to a complete eld memory structure

Description

The returned value does not include the vendor block. It's vendor specific, and comprises of the remaining bytes in the ELD memory buffer after *drm_eld_size()* bytes of header and baseline block.

The returned value is guaranteed to be a multiple of 4.

u8 **drm_eld_get_spk_alloc**(const uint8_t *eld)

Get speaker allocation

Parameters

const uint8_t *eld

pointer to an ELD memory structure

Description

The returned value is the speakers mask. User has to use DRM_ELD_SPEAKER field definitions to identify speakers.

u8 **drm_eld_get_conn_type**(const uint8_t *eld)

Get device type hdmi/dp connected

Parameters

const uint8_t *eld

pointer to an ELD memory structure

Description

The caller need to use `DRM_ELD_CONN_TYPE_HDMI` or `DRM_ELD_CONN_TYPE_DP` to identify the display type connected.

const char *drm_edid_decode_mfg_id(u16 mfg_id, char vend[4])

Decode the manufacturer ID

Parameters

u16 mfg_id

The manufacturer ID

char vend[4]

A 4-byte buffer to store the 3-letter vendor string plus a '0' termination

drm_edid_encode_panel_id

drm_edid_encode_panel_id (vend_chr_0, vend_chr_1, vend_chr_2, product_id)

Encode an ID for matching against *drm_edid_get_panel_id()*

Parameters

vend_chr_0

First character of the vendor string.

vend_chr_1

Second character of the vendor string.

vend_chr_2

Third character of the vendor string.

product_id

The 16-bit product ID.

Description

This is a macro so that it can be calculated at compile time and used as an initializer.

For instance:

`drm_edid_encode_panel_id('B', 'O', 'E', 0x2d08) => 0x09e52d08`

Return

a 32-bit ID per panel.

void drm_edid_decode_panel_id(u32 panel_id, char vend[4], u16 *product_id)

Decode a panel ID from *drm_edid_encode_panel_id()*

Parameters

u32 panel_id

The panel ID to decode.

char vend[4]

A 4-byte buffer to store the 3-letter vendor string plus a '0' termination

u16 *product_id

The product ID will be returned here.

Description

For instance, after:

```
drm_edid_decode_panel_id(0x09e52d08, vend, product_id)
```

These will be true:

```
vend[0] = 'B' vend[1] = 'O' vend[2] = 'E' vend[3] = '0' product_id = 0x2d08
```

```
int drm_edid_header_is_valid(const void *_edid)
```

sanity check the header of the base EDID block

Parameters

```
const void *_edid
```

pointer to raw base EDID block

Description

Sanity check the header of the base EDID block.

Return

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

```
bool drm_edid_are_equal(const struct edid *edid1, const struct edid *edid2)
```

compare two edid blobs.

Parameters

```
const struct edid *edid1
```

pointer to first blob

```
const struct edid *edid2
```

pointer to second blob This helper can be used during probing to determine if edid had changed.

```
bool drm_edid_block_valid(u8 *_block, int block_num, bool print_bad_edid, bool  
                          *edid_corrupt)
```

Sanity check the EDID block (base or extension)

Parameters

```
u8 *_block
```

pointer to raw EDID block

```
int block_num
```

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

```
bool print_bad_edid
```

if true, dump bad EDID blocks to the console

```
bool *edid_corrupt
```

if true, the header or checksum is invalid

Description

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

Return

True if the block is valid, false otherwise.

bool **drm_edid_is_valid**(struct *edid* *edid)
sanity check EDID data

Parameters

struct edid *edid
EDID data

Description

Sanity-check an entire EDID record (including extensions)

Return

True if the EDID data is valid, false otherwise.

bool **drm_edid_valid**(const struct *drm_edid* *drm_edid)
sanity check EDID data

Parameters

const struct drm_edid *drm_edid
EDID data

Description

Sanity check an EDID. Cross check block count against allocated size and checksum the blocks.

Return

True if the EDID data is valid, false otherwise.

int **drm_edid_override_connector_update**(struct *drm_connector* *connector)
add modes from override/firmware EDID

Parameters

struct drm_connector *connector
connector we're probing

Description

Add modes from the override/firmware EDID, if available. Only to be used from *drm_helper_probe_single_connector_modes()* as a fallback for when DDC probe failed during *drm_get_edid()* and caused the override/firmware EDID to be skipped.

Return

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

struct edid ***drm_do_get_edid**(struct *drm_connector* *connector, read_block_fn read_block,
void *context)
get EDID data using a custom EDID block read function

Parameters

struct drm_connector *connector
connector we're probing

read_block_fn read_block
EDID block read function

void *context

private data passed to the block read function

Description

When the I2C adapter connected to the DDC bus is hidden behind a device that exposes a different interface to read EDID blocks this function can be used to get EDID data using a custom block read function.

As in the general case the DDC bus is accessible by the kernel at the I2C level, drivers must make all reasonable efforts to expose it as an I2C adapter and use [drm_get_edid\(\)](#) instead of abusing this function.

The EDID may be overridden using debugfs `override_edid` or firmware EDID (`drm_edid_load_firmware()` and `drm.edid_firmware` parameter), in this priority order. Having either of them bypasses actual EDID reads.

Return

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

const struct edid ***drm_edid_raw**(const struct [drm_edid](#) *drm_edid)

Get a pointer to the raw EDID data.

Parameters

const struct **drm_edid** ***drm_edid**

drm_edid container

Description

Get a pointer to the raw EDID data.

This is for transition only. Avoid using this like the plague.

Return

Pointer to raw EDID data.

const struct **drm_edid** ***drm_edid_alloc**(const void *edid, size_t size)

Allocate a new `drm_edid` container

Parameters

const void ***edid**

Pointer to raw EDID data

size_t **size**

Size of memory allocated for EDID

Description

Allocate a new `drm_edid` container. Do not calculate edid size from edid, pass the actual size that has been allocated for the data. There is no validation of the raw EDID data against the size, but at least the EDID base block must fit in the buffer.

The returned pointer must be freed using [drm_edid_free\(\)](#).

Return

`drm_edid` container, or NULL on errors

const struct *drm_edid* ***drm_edid_dup**(const struct *drm_edid* *drm_edid)

Duplicate a *drm_edid* container

Parameters

const struct *drm_edid* ***drm_edid**

EDID to duplicate

Description

The returned pointer must be freed using *drm_edid_free()*.

Return

drm_edid container copy, or NULL on errors

void **drm_edid_free**(const struct *drm_edid* *drm_edid)

Free the *drm_edid* container

Parameters

const struct *drm_edid* ***drm_edid**

EDID to free

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

probe DDC presence

Parameters

struct *i2c_adapter* ***adapter**

I2C adapter to probe

Return

True on success, false on failure.

struct *edid* ***drm_get_edid**(struct *drm_connector* *connector, struct *i2c_adapter* *adapter)

get EDID data, if available

Parameters

struct *drm_connector* ***connector**

connector we're probing

struct *i2c_adapter* ***adapter**

I2C adapter to use for DDC

Description

Poke the given I2C channel to grab EDID data if possible. If found, attach it to the connector.

Return

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

const struct *drm_edid* ***drm_edid_read_custom**(struct *drm_connector* *connector,
read_block_fn read_block, void *context)

Read EDID data using given EDID block read function

Parameters

struct *drm_connector* ***connector**

Connector to use

read_block_fn read_block

EDID block read function

void *context

Private data passed to the block read function

Description

When the I2C adapter connected to the DDC bus is hidden behind a device that exposes a different interface to read EDID blocks this function can be used to get EDID data using a custom block read function.

As in the general case the DDC bus is accessible by the kernel at the I2C level, drivers must make all reasonable efforts to expose it as an I2C adapter and use [drm_edid_read\(\)](#) or [drm_edid_read_ddc\(\)](#) instead of abusing this function.

The EDID may be overridden using debugfs `override_edid` or firmware EDID (`drm_edid_load_firmware()` and `drm.edid_firmware` parameter), in this priority order. Having either of them bypasses actual EDID reads.

The returned pointer must be freed using [drm_edid_free\(\)](#).

Return

Pointer to EDID, or NULL if probe/read failed.

```
const struct drm_edid *drm_edid_read_ddc(struct drm\_connector *connector, struct  
                                         i2c_adapter *adapter)
```

Read EDID data using given I2C adapter

Parameters

struct drm_connector *connector

Connector to use

struct i2c_adapter *adapter

I2C adapter to use for DDC

Description

Read EDID using the given I2C adapter.

The EDID may be overridden using debugfs `override_edid` or firmware EDID (`drm_edid_load_firmware()` and `drm.edid_firmware` parameter), in this priority order. Having either of them bypasses actual EDID reads.

Prefer initializing `connector->ddc` with [drm_connector_init_with_ddc\(\)](#) and using [drm_edid_read\(\)](#) instead of this function.

The returned pointer must be freed using [drm_edid_free\(\)](#).

Return

Pointer to EDID, or NULL if probe/read failed.

```
const struct drm_edid *drm_edid_read(struct drm\_connector *connector)
```

Read EDID data using connector's I2C adapter

Parameters

struct drm_connector *connector

Connector to use

Description

Read EDID using the connector's I2C adapter.

The EDID may be overridden using `debugfs override_edid` or firmware EDID (`drm_edid_load_firmware()` and `drm.edid_firmware` parameter), in this priority order. Having either of them bypasses actual EDID reads.

The returned pointer must be freed using `drm_edid_free()`.

Return

Pointer to EDID, or NULL if probe/read failed.

u32 **drm_edid_get_panel_id**(struct i2c_adapter *adapter)

Get a panel's ID through DDC

Parameters

struct i2c_adapter *adapter

I2C adapter to use for DDC

Description

This function reads the first block of the EDID of a panel and (assuming that the EDID is valid) extracts the ID out of it. The ID is a 32-bit value (16 bits of manufacturer ID and 16 bits of per-manufacturer ID) that's supposed to be different for each different model of panel.

This function is intended to be used during early probing on devices where more than one panel might be present. Because of its intended use it must assume that the EDID of the panel is correct, at least as far as the ID is concerned (in other words, we don't process any overrides here).

NOTE

it's expected that this function and `drm_do_get_edid()` will both be read the EDID, but there is no caching between them. Since we're only reading the first block, hopefully this extra overhead won't be too big.

Return

A 32-bit ID that should be different for each make/model of panel.

See the functions `drm_edid_encode_panel_id()` and `drm_edid_decode_panel_id()` for some details on the structure of this ID.

struct edid ***drm_get_edid_switcheroo**(struct *drm_connector* *connector, struct i2c_adapter *adapter)

get EDID data for a vga_switcheroo output

Parameters

struct drm_connector *connector

connector we're probing

struct i2c_adapter *adapter

I2C adapter to use for DDC

Description

Wrapper around `drm_get_edid()` for laptops with dual GPUs using one set of outputs. The wrapper adds the requisite `vga_switcheroo` calls to temporarily switch DDC to the GPU which is retrieving EDID.

Return

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

```
const struct drm_edid *drm_edid_read_switcheroo(struct drm_connector *connector, struct
                                                i2c_adapter *adapter)

    get EDID data for a vga_switcheroo output
```

Parameters

```
struct drm_connector *connector
    connector we're probing

struct i2c_adapter *adapter
    I2C adapter to use for DDC
```

Description

Wrapper around `drm_edid_read_ddc()` for laptops with dual GPUs using one set of outputs. The wrapper adds the requisite `vga_switcheroo` calls to temporarily switch DDC to the GPU which is retrieving EDID.

Return

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

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

    duplicate an EDID and the extensions
```

Parameters

```
const struct edid *edid
    EDID to duplicate
```

Return

Pointer to duplicated EDID or NULL on allocation failure.

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

    look for a CEA mode matching given mode
```

Parameters

```
const struct drm_display_mode *to_match
    display mode
```

Return

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

```
struct drm_display_mode *drm_display_mode_from_cea_vic(struct drm_device *dev, u8
                                                         video_code)

    return a mode for CEA VIC
```

Parameters

```
struct drm_device *dev
    DRM device
```

u8 video_code

CEA VIC of the mode

Description

Creates a new mode matching the specified CEA VIC.

Return

A new `drm_display_mode` on success or `NULL` on failure

void **drm_edid_get_monitor_name**(const struct *edid* *edid, char *name, int bufsize)
fetch the monitor name from the edid

Parameters

const struct edid *edid
monitor EDID information

char *name
pointer to a character array to hold the name of the monitor

int bufsize
The size of the name buffer (should be at least 14 chars.)

int **drm_edid_to_sad**(const struct *edid* *edid, struct cea_sad **sads)
extracts SADs from EDID

Parameters

const struct edid *edid
EDID to parse

struct cea_sad **sads
pointer that will be set to the extracted SADs

Description

Looks for CEA EDID block and extracts SADs (Short Audio Descriptors) from it.

Note

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

Return

The number of found SADs or negative number on error.

int **drm_edid_to_speaker_allocation**(const struct *edid* *edid, u8 **sadb)
extracts Speaker Allocation Data Blocks from EDID

Parameters

const struct edid *edid
EDID to parse

u8 **sadb
pointer to the speaker block

Description

Looks for CEA EDID block and extracts the Speaker Allocation Data Block from it.

Note

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

Return

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

int **drm_av_sync_delay**(struct *drm_connector* *connector, const struct *drm_display_mode* *mode)

compute the HDMI/DP sink audio-video sync delay

Parameters

struct drm_connector *connector

connector associated with the HDMI/DP sink

const struct drm_display_mode *mode

the display mode

Return

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

bool **drm_detect_hdmi_monitor**(const struct *edid* *edid)

detect whether monitor is HDMI

Parameters

const struct edid *edid

monitor EDID information

Description

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

Drivers that have added the modes parsed from EDID to `drm_display_info` should use *drm_display_info.is_hdmi* instead of calling this function.

Return

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

bool **drm_detect_monitor_audio**(const struct *edid* *edid)

check monitor audio capability

Parameters

const struct edid *edid

EDID block to scan

Description

Monitor should have CEA extension block. If monitor has 'basic audio', but no CEA audio blocks, it's 'basic audio' only. If there is any audio extension block and supported audio format, assume at least 'basic audio' support, even if 'basic audio' is not defined in EDID.

Return

True if the monitor supports audio, false otherwise.

enum hdmi_quantization_range **drm_default_rgb_quant_range**(const struct *drm_display_mode* *mode)
default RGB quantization range

Parameters

const struct drm_display_mode *mode
display mode

Description

Determine the default RGB quantization range for the mode, as specified in CEA-861.

Return

The default RGB quantization range for the mode

int **drm_edid_connector_update**(struct *drm_connector* *connector, const struct *drm_edid* *drm_edid)

Update connector information from EDID

Parameters

struct drm_connector *connector
Connector

const struct drm_edid *drm_edid
EDID

Description

Update the connector display info, ELD, HDR metadata, relevant properties, etc. from the passed in EDID.

If EDID is NULL, reset the information.

Must be called before calling *drm_edid_connector_add_modes()*.

Return

0 on success, negative error on errors.

int **drm_edid_connector_add_modes**(struct *drm_connector* *connector)
Update probed modes from the EDID property

Parameters

struct drm_connector *connector
Connector

Description

Add the modes from the previously updated EDID property to the connector probed modes list. *drm_edid_connector_update()* must have been called before this to update the EDID property.

Return

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

int **drm_connector_update_edid_property**(struct *drm_connector* *connector, const struct *edid* *edid)
update the edid property of a connector

Parameters

struct drm_connector *connector
drm connector

const struct edid *edid
new value of the edid property

Description

This function creates a new blob modeset object and assigns its id to the connector's edid property. Since we also parse tile information from EDID's displayID block, we also set the connector's tile property here. See [drm_connector_set_tile_property\(\)](#) for more details.

This function is deprecated. Use [drm_edid_connector_update\(\)](#) instead.

Return

Zero on success, negative errno on failure.

int **drm_add_edid_modes**(struct [drm_connector](#) *connector, struct [edid](#) *edid)
add modes from EDID data, if available

Parameters

struct drm_connector *connector
connector we're probing

struct edid *edid
EDID data

Description

Add the specified modes to the connector's mode list. Also fills out the [drm_display_info](#) structure and ELD in **connector** with any information which can be derived from the edid.

This function is deprecated. Use [drm_edid_connector_add_modes\(\)](#) instead.

Return

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

int **drm_add_modes_noedid**(struct [drm_connector](#) *connector, int hdisplay, int vdisplay)
add modes for the connectors without EDID

Parameters

struct drm_connector *connector
connector we're probing

int hdisplay
the horizontal display limit

int vdisplay
the vertical display limit

Description

Add the specified modes to the connector's mode list. Only when the hdisplay/vdisplay is not beyond the given limit, it will be added.

Return

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

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

Sets the preferred mode of a connector

Parameters

```
struct drm_connector *connector
```

connector whose mode list should be processed

```
int hpref
```

horizontal resolution of preferred mode

```
int vpref
```

vertical resolution of preferred mode

Description

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

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

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

Parameters

```
struct hdmi_avi_infoframe *frame
```

HDMI AVI infoframe

```
const struct drm_connector *connector
```

the connector

```
const struct drm_display_mode *mode
```

DRM display mode

Return

0 on success or a negative error code on failure.

```
void drm_hdmi_avi_infoframe_quant_range(struct hdmi_avi_infoframe *frame, const struct  
                                         drm_connector *connector, const struct  
                                         drm_display_mode *mode, enum  
                                         hdmi_quantization_range rgb_quant_range)
```

fill the HDMI AVI infoframe quantization range information

Parameters

```
struct hdmi_avi_infoframe *frame
```

HDMI AVI infoframe

```
const struct drm_connector *connector
```

the connector

```
const struct drm_display_mode *mode
```

DRM display mode

```
enum hdmi_quantization_range rgb_quant_range
```

RGB quantization range (Q)

```
int drm_hdmi_vendor_infoframe_from_display_mode(struct hdmi_vendor_infoframe *frame,
                                                const struct drm_connector
                                                *connector, const struct
                                                drm_display_mode *mode)
```

fill an HDMI infoframe with data from a DRM display mode

Parameters

struct hdmi_vendor_infoframe *frame
HDMI vendor infoframe

const struct drm_connector *connector
the connector

const struct drm_display_mode *mode
DRM display mode

Description

Note that there's a need to send HDMI vendor infoframes only when using a 4k or stereoscopic 3D mode. So when giving any other mode as input this function will return -EINVAL, error that can be safely ignored.

Return

0 on success or a negative error code on failure.

5.21 SCDC Helper Functions Reference

Status and Control Data Channel (SCDC) is a mechanism introduced by the HDMI 2.0 specification. It is a point-to-point protocol that allows the HDMI source and HDMI sink to exchange data. The same I2C interface that is used to access EDID serves as the transport mechanism for SCDC.

Note: The SCDC status is going to be lost when the display is disconnected. This can happen physically when the user disconnects the cable, but also when a display is switched on (such as waking up a TV).

This is further complicated by the fact that, upon a disconnection / reconnection, KMS won't change the mode on its own. This means that one can't just rely on setting the SCDC status on enable, but also has to track the connector status changes using interrupts and restore the SCDC status. The typical solution for this is to trigger an empty modeset in `drm_connector_helper_funcs.detect_ctx()`, like what `vc4` does in `vc4_hdmi_reset_link()`.

```
int drm_scdc_readb(struct i2c_adapter *adapter, u8 offset, u8 *value)
    read a single byte from SCDC
```

Parameters

struct i2c_adapter *adapter
I2C adapter

u8 offset
offset of register to read

u8 *value
return location for the register value

Description

Reads a single byte from SCDC. This is a convenience wrapper around the [drm_scdc_read\(\)](#) function.

Return

0 on success or a negative error code on failure.

int **drm_scdc_writeb**(struct i2c_adapter *adapter, u8 offset, u8 value)
 write a single byte to SCDC

Parameters

struct i2c_adapter *adapter
 I2C adapter

u8 offset
 offset of register to read

u8 value
 return location for the register value

Description

Writes a single byte to SCDC. This is a convenience wrapper around the [drm_scdc_write\(\)](#) function.

Return

0 on success or a negative error code on failure.

ssize_t **drm_scdc_read**(struct i2c_adapter *adapter, u8 offset, void *buffer, size_t size)
 read a block of data from SCDC

Parameters

struct i2c_adapter *adapter
 I2C controller

u8 offset
 start offset of block to read

void *buffer
 return location for the block to read

size_t size
 size of the block to read

Description

Reads a block of data from SCDC, starting at a given offset.

Return

0 on success, negative error code on failure.

ssize_t **drm_scdc_write**(struct i2c_adapter *adapter, u8 offset, const void *buffer, size_t size)
 write a block of data to SCDC

Parameters

struct i2c_adapter *adapter
I2C controller

u8 offset
start offset of block to write

const void *buffer
block of data to write

size_t size
size of the block to write

Description

Writes a block of data to SCDC, starting at a given offset.

Return

0 on success, negative error code on failure.

bool **drm_scdc_get_scrambling_status**(struct *drm_connector* *connector)
what is status of scrambling?

Parameters

struct drm_connector *connector
connector

Description

Reads the scrambler status over SCDC, and checks the scrambling status.

Return

True if the scrambling is enabled, false otherwise.

bool **drm_scdc_set_scrambling**(struct *drm_connector* *connector, bool enable)
enable scrambling

Parameters

struct drm_connector *connector
connector

bool enable
bool to indicate if scrambling is to be enabled/disabled

Description

Writes the TMDS config register over SCDC channel, and: enables scrambling when enable = 1 disables scrambling when enable = 0

Return

True if scrambling is set/reset successfully, false otherwise.

bool **drm_scdc_set_high_tmds_clock_ratio**(struct *drm_connector* *connector, bool set)
set TMDS clock ratio

Parameters

struct drm_connector *connector
connector

bool set

ret or reset the high clock ratio

TMDS clock ratio calculations go like this:

TMDS character = 10 bit TMDS encoded value

TMDS character rate = The rate at which TMDS characters are transmitted (Mcsc)

TMDS bit rate = 10x TMDS character rate

As per the spec:

TMDS clock rate for pixel clock < 340 MHz = 1x the character rate = 1/10 pixel clock rate

TMDS clock rate for pixel clock > 340 MHz = 0.25x the character rate = 1/40 pixel clock rate

Writes to the TMDS config register over SCDC channel, and:

sets TMDS clock ratio to 1/40 when set = 1

sets TMDS clock ratio to 1/10 when set = 0

Return

True if write is successful, false otherwise.

5.22 HDMI Infoframes Helper Reference

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

struct `hdr_sink_metadata`

HDR sink metadata

Definition:

```
struct hdr_sink_metadata {
    __u32 metadata_type;
    union {
        struct hdr_static_metadata hdmi_type1;
    };
};
```

Members**`metadata_type`**

Static_Metadata_Descriptor_ID.

`{unnamed_union}`

anonymous

`hdmi_type1`

HDR Metadata Infoframe.

Description

Metadata Information read from Sink's EDID

union **hdmi_infoframe**

overall union of all abstract infoframe representations

Definition:

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

Members

any

generic infoframe

avi

avi infoframe

spd

spd infoframe

vendor

union of all vendor infoframes

audio

audio infoframe

drm

Dynamic Range and Mastering infoframe

Description

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

void **hdmi_avi_infoframe_init**(struct hdmi_avi_infoframe *frame)

initialize an HDMI AVI infoframe

Parameters

struct **hdmi_avi_infoframe** *frame

HDMI AVI infoframe

int **hdmi_avi_infoframe_check**(struct hdmi_avi_infoframe *frame)

check a HDMI AVI infoframe

Parameters

struct **hdmi_avi_infoframe** *frame

HDMI AVI infoframe

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields.

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

`ssize_t hdmi_avi_infoframe_pack_only(const struct hdmi_avi_infoframe *frame, void *buffer, size_t size)`

write HDMI AVI infoframe to binary buffer

Parameters

const struct hdmi_avi_infoframe *frame
HDMI AVI infoframe

void *buffer
destination buffer

size_t size
size of buffer

Description

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

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

`ssize_t hdmi_avi_infoframe_pack(struct hdmi_avi_infoframe *frame, void *buffer, size_t size)`
check a HDMI AVI infoframe, and write it to binary buffer

Parameters

struct hdmi_avi_infoframe *frame
HDMI AVI infoframe

void *buffer
destination buffer

size_t size
size of buffer

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields, after which it packs the information contained in the **frame** structure into a binary representation that can be written into the corresponding controller registers. This function also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

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

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

initialize an HDMI SPD infoframe

Parameters

struct hdmi_spd_infoframe *frame
HDMI SPD infoframe

const char *vendor
vendor string

const char *product
product string

Description

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

int **hdmi_spd_infoframe_check**(struct hdmi_spd_infoframe *frame)
check a HDMI SPD infoframe

Parameters

struct hdmi_spd_infoframe *frame
HDMI SPD infoframe

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields.

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

ssize_t **hdmi_spd_infoframe_pack_only**(const struct hdmi_spd_infoframe *frame, void *buffer, size_t size)
write HDMI SPD infoframe to binary buffer

Parameters

const struct hdmi_spd_infoframe *frame
HDMI SPD infoframe

void *buffer
destination buffer

size_t size
size of buffer

Description

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

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

ssize_t **hdmi_spd_infoframe_pack**(struct hdmi_spd_infoframe *frame, void *buffer, size_t size)
check a HDMI SPD infoframe, and write it to binary buffer

Parameters

struct hdmi_spd_infoframe *frame
HDMI SPD infoframe

void *buffer
destination buffer

size_t size
size of buffer

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields, after which it packs the information contained in the **frame** structure into a binary representation that can be written into the corresponding controller registers. This function also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

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

```
int hdmi_audio_infoframe_init(struct hdmi_audio_infoframe *frame)  
    initialize an HDMI audio infoframe
```

Parameters

```
struct hdmi_audio_infoframe *frame  
    HDMI audio infoframe
```

Description

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

```
int hdmi_audio_infoframe_check(const struct hdmi_audio_infoframe *frame)  
    check a HDMI audio infoframe
```

Parameters

```
const struct hdmi_audio_infoframe *frame  
    HDMI audio infoframe
```

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields.

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

```
ssize_t hdmi_audio_infoframe_pack_only(const struct hdmi_audio_infoframe *frame, void  
                                         *buffer, size_t size)  
    write HDMI audio infoframe to binary buffer
```

Parameters

```
const struct hdmi_audio_infoframe *frame  
    HDMI audio infoframe
```

```
void *buffer  
    destination buffer
```

```
size_t size  
    size of buffer
```

Description

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

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

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

check a HDMI Audio infoframe, and write it to binary buffer

Parameters

struct hdmi_audio_infoframe *frame
HDMI Audio infoframe

void *buffer
destination buffer

size_t size
size of buffer

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields, after which it packs the information contained in the **frame** structure into a binary representation that can be written into the corresponding controller registers. This function also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

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

ssize_t **hdmi_audio_infoframe_pack_for_dp**(const struct hdmi_audio_infoframe *frame, struct *dp_sdp* *sdp, u8 dp_version)

Pack a HDMI Audio infoframe for DisplayPort

Parameters

const struct hdmi_audio_infoframe *frame
HDMI Audio infoframe

struct dp_sdp *sdp
Secondary data packet for DisplayPort.

u8 dp_version
DisplayPort version to be encoded in the header

Description

Packs a HDMI Audio Infoframe to be sent over DisplayPort. This function fills the secondary data packet to be used for DisplayPort.

Return

Number of total written bytes or a negative errno on failure.

int **hdmi_vendor_infoframe_init**(struct hdmi_vendor_infoframe *frame)
initialize an HDMI vendor infoframe

Parameters

struct hdmi_vendor_infoframe *frame
HDMI vendor infoframe

Description

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

int **hdmi_vendor_infoframe_check**(struct hdmi_vendor_infoframe *frame)

check a HDMI vendor infoframe

Parameters

struct hdmi_vendor_infoframe *frame

HDMI infoframe

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields.

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

ssize_t **hdmi_vendor_infoframe_pack_only**(const struct hdmi_vendor_infoframe *frame, void *buffer, size_t size)

write a HDMI vendor infoframe to binary buffer

Parameters

const struct hdmi_vendor_infoframe *frame

HDMI infoframe

void *buffer

destination buffer

size_t size

size of buffer

Description

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

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

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

check a HDMI Vendor infoframe, and write it to binary buffer

Parameters

struct hdmi_vendor_infoframe *frame

HDMI Vendor infoframe

void *buffer

destination buffer

size_t size

size of buffer

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields, after which it packs the information contained in the **frame** structure into a binary representation that can be written into the corresponding controller registers. This function also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

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

int **hdmi_drm_infoframe_init**(struct hdmi_drm_infoframe *frame)
initialize an HDMI Dynaminc Range and mastering infoframe

Parameters

struct hdmi_drm_infoframe *frame
HDMI DRM infoframe

Description

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

int **hdmi_drm_infoframe_check**(struct hdmi_drm_infoframe *frame)
check a HDMI DRM infoframe

Parameters

struct hdmi_drm_infoframe *frame
HDMI DRM infoframe

Description

Validates that the infoframe is consistent. Returns 0 on success or a negative error code on failure.

ssize_t **hdmi_drm_infoframe_pack_only**(const struct hdmi_drm_infoframe *frame, void *buffer, size_t size)
write HDMI DRM infoframe to binary buffer

Parameters

const struct hdmi_drm_infoframe *frame
HDMI DRM infoframe

void *buffer
destination buffer

size_t size
size of buffer

Description

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

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

ssize_t **hdmi_drm_infoframe_pack**(struct hdmi_drm_infoframe *frame, void *buffer, size_t size)
check a HDMI DRM infoframe, and write it to binary buffer

Parameters

struct hdmi_drm_infoframe *frame
HDMI DRM infoframe

void *buffer
destination buffer

size_t size

size of buffer

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields, after which it packs the information contained in the **frame** structure into a binary representation that can be written into the corresponding controller registers. This function also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

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

int **hdmi_infoframe_check**(union *hdmi_infoframe* *frame)

check a HDMI infoframe

Parameters**union** **hdmi_infoframe** *frame

HDMI infoframe

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields.

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

ssize_t **hdmi_infoframe_pack_only**(const union *hdmi_infoframe* *frame, void *buffer, size_t size)

write a HDMI infoframe to binary buffer

Parameters**const** union **hdmi_infoframe** *frame

HDMI infoframe

void *buffer

destination buffer

size_t size

size of buffer

Description

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

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

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

check a HDMI infoframe, and write it to binary buffer

Parameters**union** **hdmi_infoframe** *frame

HDMI infoframe

void *buffer

destination buffer

size_t size

size of buffer

Description

Validates that the infoframe is consistent and updates derived fields (eg. length) based on other fields, after which it packs the information contained in the **frame** structure into a binary representation that can be written into the corresponding controller registers. This function also computes the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

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

void **hdmi_infoframe_log**(const char *level, struct device *dev, const union *hdmi_infoframe* *frame)

log info of HDMI infoframe

Parameters

const char *level

logging level

struct device *dev

device

const union hdmi_infoframe *frame

HDMI infoframe

int **hdmi_drm_infoframe_unpack_only**(struct hdmi_drm_infoframe *frame, const void *buffer, size_t size)

unpack binary buffer of CTA-861-G DRM infoframe DataBytes to a HDMI DRM infoframe

Parameters

struct hdmi_drm_infoframe *frame

HDMI DRM infoframe

const void *buffer

source buffer

size_t size

size of buffer

Description

Unpacks CTA-861-G DRM infoframe DataBytes contained in the binary **buffer** into a structured **frame** of the HDMI Dynamic Range and Mastering (DRM) infoframe.

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

int **hdmi_infoframe_unpack**(union *hdmi_infoframe* *frame, const void *buffer, size_t size)

unpack binary buffer to a HDMI infoframe

Parameters

union hdmi_infoframe *frame

HDMI infoframe

const void *buffer

source buffer

size_t size

size of buffer

Description

Unpacks the information contained in binary buffer **buffer** into a structured **frame** of a HDMI infoframe. Also verifies the checksum as required by section 5.3.5 of the HDMI 1.4 specification.

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

5.23 Rectangle Utilities Reference

Utility functions to help manage rectangular areas for clipping, scaling, etc. calculations.

struct **drm_rect**

two dimensional rectangle

Definition:

```
struct drm_rect {  
    int x1, y1, x2, y2;  
};
```

Members

x1

horizontal starting coordinate (inclusive)

y1

vertical starting coordinate (inclusive)

x2

horizontal ending coordinate (exclusive)

y2

vertical ending coordinate (exclusive)

Description

Note that this must match the layout of *struct drm_mode_rect* or the damage helpers like *drm_atomic_helper_damage_iter_init()* break.

DRM_RECT_INIT

DRM_RECT_INIT (x, y, w, h)

initialize a rectangle from x/y/w/h

Parameters

x

x coordinate

y

y coordinate

w

width

h

height

Return

A new rectangle of the specified size.

DRM_RECT_FMT

DRM_RECT_FMT ()

printf string for *struct drm_rect*

Parameters

DRM_RECT_ARG

DRM_RECT_ARG (r)

printf arguments for *struct drm_rect*

Parameters

r

rectangle struct

DRM_RECT_FP_FMT

DRM_RECT_FP_FMT ()

printf string for *struct drm_rect* in 16.16 fixed point

Parameters

DRM_RECT_FP_ARG

DRM_RECT_FP_ARG (r)

printf arguments for *struct drm_rect* in 16.16 fixed point

Parameters

r

rectangle struct

Description

This is useful for e.g. printing plane source rectangles, which are in 16.16 fixed point.

void **drm_rect_init**(struct *drm_rect* *r, int x, int y, int width, int height)

initialize the rectangle from x/y/w/h

Parameters

struct drm_rect *r

rectangle

int x

x coordinate

int y

y coordinate

int width
width

int height
height

void **drm_rect_adjust_size**(struct *drm_rect* *r, int dw, int dh)
adjust the size of the rectangle

Parameters

struct drm_rect *r
rectangle to be adjusted

int dw
horizontal adjustment

int dh
vertical adjustment

Description

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

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

void **drm_rect_translate**(struct *drm_rect* *r, int dx, int dy)
translate the rectangle

Parameters

struct drm_rect *r
rectangle to be translated

int dx
horizontal translation

int dy
vertical translation

Description

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

void **drm_rect_translate_to**(struct *drm_rect* *r, int x, int y)
translate the rectangle to an absolute position

Parameters

struct drm_rect *r
rectangle to be translated

int x
horizontal position

int y
vertical position

Description

Move rectangle **r** to **x** in the horizontal direction, and to **y** in the vertical direction.

void **drm_rect_downscale**(struct *drm_rect* *r, int horz, int vert)
 downscale a rectangle

Parameters

struct drm_rect *r
 rectangle to be downscaled

int horz
 horizontal downscale factor

int vert
 vertical downscale factor

Description

Divide the coordinates of rectangle **r** by **horz** and **vert**.

int **drm_rect_width**(const struct *drm_rect* *r)
 determine the rectangle width

Parameters

const struct drm_rect *r
 rectangle whose width is returned

Return

The width of the rectangle.

int **drm_rect_height**(const struct *drm_rect* *r)
 determine the rectangle height

Parameters

const struct drm_rect *r
 rectangle whose height is returned

Return

The height of the rectangle.

bool **drm_rect_visible**(const struct *drm_rect* *r)
 determine if the rectangle is visible

Parameters

const struct drm_rect *r
 rectangle whose visibility is returned

Return

true if the rectangle is visible, false otherwise.

bool **drm_rect_equals**(const struct *drm_rect* *r1, const struct *drm_rect* *r2)
 determine if two rectangles are equal

Parameters

const struct drm_rect *r1
 first rectangle

const struct drm_rect *r2
second rectangle

Return

true if the rectangles are equal, false otherwise.

void **drm_rect_fp_to_int**(struct *drm_rect* *dst, const struct *drm_rect* *src)
Convert a rect in 16.16 fixed point form to int form.

Parameters

struct drm_rect *dst
rect to be stored the converted value

const struct drm_rect *src
rect in 16.16 fixed point form

bool **drm_rect_intersect**(struct *drm_rect* *r1, const struct *drm_rect* *r2)
intersect two rectangles

Parameters

struct drm_rect *r1
first rectangle

const struct drm_rect *r2
second rectangle

Description

Calculate the intersection of rectangles **r1** and **r2**. **r1** will be overwritten with the intersection.

Return

true if rectangle **r1** is still visible after the operation, false otherwise.

bool **drm_rect_clip_scaled**(struct *drm_rect* *src, struct *drm_rect* *dst, const struct *drm_rect* *clip)
perform a scaled clip operation

Parameters

struct drm_rect *src
source window rectangle

struct drm_rect *dst
destination window rectangle

const struct drm_rect *clip
clip rectangle

Description

Clip rectangle **dst** by rectangle **clip**. Clip rectangle **src** by the corresponding amounts, retaining the vertical and horizontal scaling factors from **src** to **dst**.

true if rectangle **dst** is still visible after being clipped, false otherwise.

Return

int **drm_rect_calc_hscale**(const struct *drm_rect* *src, const struct *drm_rect* *dst, int min_hscale, int max_hscale)

calculate the horizontal scaling factor

Parameters

const struct *drm_rect* *src

source window rectangle

const struct *drm_rect* *dst

destination window rectangle

int min_hscale

minimum allowed horizontal scaling factor

int max_hscale

maximum allowed horizontal scaling factor

Description

Calculate the horizontal scaling factor as (**src** width) / (**dst** width).

If the scale is below 1 << 16, round down. If the scale is above 1 << 16, round up. This will calculate the scale with the most pessimistic limit calculation.

Return

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

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

calculate the vertical scaling factor

Parameters

const struct *drm_rect* *src

source window rectangle

const struct *drm_rect* *dst

destination window rectangle

int min_vscale

minimum allowed vertical scaling factor

int max_vscale

maximum allowed vertical scaling factor

Description

Calculate the vertical scaling factor as (**src** height) / (**dst** height).

If the scale is below 1 << 16, round down. If the scale is above 1 << 16, round up. This will calculate the scale with the most pessimistic limit calculation.

Return

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

void **drm_rect_debug_print**(const char *prefix, const struct *drm_rect* *r, bool fixed_point)
print the rectangle information

Parameters

const char *prefix

prefix string

const struct drm_rect *r

rectangle to print

bool fixed_point

rectangle is in 16.16 fixed point format

void **drm_rect_rotate**(struct *drm_rect* *r, int width, int height, unsigned int rotation)

Rotate the rectangle

Parameters

struct drm_rect *r

rectangle to be rotated

int width

Width of the coordinate space

int height

Height of the coordinate space

unsigned int rotation

Transformation to be applied

Description

Apply **rotation** to the coordinates of rectangle **r**.

width and **height** combined with **rotation** define the location of the new origin.

width corresponds to the horizontal and **height** to the vertical axis of the untransformed coordinate space.

void **drm_rect_rotate_inv**(struct *drm_rect* *r, int width, int height, unsigned int rotation)

Inverse rotate the rectangle

Parameters

struct drm_rect *r

rectangle to be rotated

int width

Width of the coordinate space

int height

Height of the coordinate space

unsigned int rotation

Transformation whose inverse is to be applied

Description

Apply the inverse of **rotation** to the coordinates of rectangle **r**.

width and **height** combined with **rotation** define the location of the new origin.

width corresponds to the horizontal and **height** to the vertical axis of the original untransformed coordinate space, so that you never have to flip them when doing a rotation and its inverse. That is, if you do

```
drm_rect_rotate(&r, width, height, rotation);
drm_rect_rotate_inv(&r, width, height, rotation);
```

you will always get back the original rectangle.

5.24 Flip-work Helper Reference

Util to queue up work to run from work-queue context after flip/vblank. Typically this can be used to defer unref of framebuffer's, cursor bo's, etc until after vblank. The APIs are all thread-safe. Moreover, `drm_flip_work_queue_task` and `drm_flip_work_queue` can be called in atomic context.

struct **drm_flip_task**
flip work task

Definition:

```
struct drm_flip_task {
    struct list_head node;
    void *data;
};
```

Members

node
list entry element

data
data to pass to *drm_flip_work.func*

struct **drm_flip_work**
flip work queue

Definition:

```
struct drm_flip_work {
    const char *name;
    drm_flip_func_t func;
    struct work_struct worker;
    struct list_head queued;
    struct list_head committed;
    spinlock_t lock;
};
```

Members

name
debug name

func
callback fxn called for each committed item

worker
worker which calls **func**

queued

queued tasks

committed

committed tasks

lock

lock to access queued and committed lists

struct *drm_flip_task* ***drm_flip_work_allocate_task**(void *data, gfp_t flags)

allocate a flip-work task

Parameters**void *data**

data associated to the task

gfp_t flags

allocator flags

Description

Allocate a *drm_flip_task* object and attach private data to it.

void **drm_flip_work_queue_task**(struct *drm_flip_work* *work, struct *drm_flip_task* *task)

queue a specific task

Parameters**struct *drm_flip_work* *work**

the flip-work

struct *drm_flip_task* *task

the task to handle

Description

Queues task, that will later be run (passed back to *drm_flip_func_t* func) on a work queue after *drm_flip_work_commit()* is called.

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

queue work

Parameters**struct *drm_flip_work* *work**

the flip-work

void *val

the value to queue

Description

Queues work, that will later be run (passed back to *drm_flip_func_t* func) on a work queue after *drm_flip_work_commit()* is called.

void **drm_flip_work_commit**(struct *drm_flip_work* *work, struct *workqueue_struct* *wq)

commit queued work

Parameters

struct **drm_flip_work** *work
the flip-work

struct **workqueue_struct** *wq
the work-queue to run the queued work on

Description

Trigger work previously queued by *drm_flip_work_queue()* to run on a workqueue. The typical usage would be to queue work (via *drm_flip_work_queue()*) at any point (from vblank irq and/or prior), and then from vblank irq commit the queued work.

void **drm_flip_work_init**(struct *drm_flip_work* *work, const char *name, drm_flip_func_t func)
initialize flip-work

Parameters

struct **drm_flip_work** *work
the flip-work to initialize

const **char** *name
debug name

drm_flip_func_t func
the callback work function

Description

Initializes/allocates resources for the flip-work

void **drm_flip_work_cleanup**(struct *drm_flip_work* *work)
cleans up flip-work

Parameters

struct **drm_flip_work** *work
the flip-work to cleanup

Description

Destroy resources allocated for the flip-work

5.25 Auxiliary Modeset Helpers

This helper library contains various one-off functions which don't really fit anywhere else in the DRM modeset helper library.

void **drm_helper_move_panel_connectors_to_head**(struct *drm_device* *dev)
move panels to the front in the connector list

Parameters

struct **drm_device** *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/DSI) panels to the front of the connector list, instead of painstakingly trying to initialize them in the right order.

```
void drm_helper_mode_fill_fb_struct(struct drm_device *dev, struct drm_framebuffer *fb,  
                                   const struct drm_mode_fb_cmd2 *mode_cmd)
```

fill out framebuffer metadata

Parameters

struct *drm_device* *dev
DRM device

struct *drm_framebuffer* *fb
drm_framebuffer object to fill out

const struct *drm_mode_fb_cmd2* *mode_cmd
metadata from the userspace fb creation request

Description

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

```
int drm_crtc_init(struct drm_device *dev, struct drm_crtc *crtc, const struct drm_crtc_funcs  
                 *funcs)
```

Legacy CRTC initialization function

Parameters

struct *drm_device* *dev
DRM device

struct *drm_crtc* *crtc
CRTC object to init

const struct *drm_crtc_funcs* *funcs
callbacks for the new CRTC

Description

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

Note that we make some assumptions about hardware limitations that may not be true for all hardware:

1. Primary plane cannot be repositioned.
2. Primary plane cannot be scaled.
3. Primary plane must cover the entire CRTC.
4. Subpixel positioning is not supported.
5. The primary plane must always be on if the CRTC is enabled.

This is purely a backwards compatibility helper for old drivers. Drivers should instead implement their own primary plane. Atomic drivers must do so. Drivers with the above hardware restriction can look into using *struct *drm_simple_display_pipe**, which encapsulates the above limitations into a nice interface.

Return

Zero on success, error code on failure.

int **drm_mode_config_helper_suspend**(struct *drm_device* *dev)
Modeset suspend helper

Parameters

struct drm_device *dev
DRM device

Description

This helper function takes care of suspending the modeset side. It disables output polling if initialized, suspends fbdev if used and finally calls *drm_atomic_helper_suspend()*. If suspending fails, fbdev and polling is re-enabled.

See also: *drm_kms_helper_poll_disable()* and *drm_fb_helper_set_suspend_unlocked()*.

Return

Zero on success, negative error code on error.

int **drm_mode_config_helper_resume**(struct *drm_device* *dev)
Modeset resume helper

Parameters

struct drm_device *dev
DRM device

Description

This helper function takes care of resuming the modeset side. It calls *drm_atomic_helper_resume()*, resumes fbdev if used and enables output polling if initialized.

See also: *drm_fb_helper_set_suspend_unlocked()* and *drm_kms_helper_poll_enable()*.

Return

Zero on success, negative error code on error.

5.26 OF/DT Helpers

A set of helper functions to aid DRM drivers in parsing standard DT properties.

uint32_t **drm_of_crtc_port_mask**(struct *drm_device* *dev, struct device_node *port)
find the mask of a registered CRTC by port OF node

Parameters

struct drm_device *dev
DRM device

struct device_node *port
port OF node

Description

Given a port OF node, return the possible mask of the corresponding CRTC within a device's list of CRTCs. Returns zero if not found.

`uint32_t drm_of_find_possible_crtcs(struct drm_device *dev, struct device_node *port)`
find the possible CRTCs for an encoder port

Parameters

`struct drm_device *dev`
DRM device

`struct device_node *port`
encoder port to scan for endpoints

Description

Scan all endpoints attached to a port, locate their attached CRTCs, and generate the DRM mask of CRTCs which may be attached to this encoder.

See Documentation/devicetree/bindings/graph.txt for the bindings.

`void drm_of_component_match_add(struct device *master, struct component_match **matchptr, int (*compare)(struct device*, void*), struct device_node *node)`

Add a component helper OF node match rule

Parameters

`struct device *master`
master device

`struct component_match **matchptr`
component match pointer

`int (*compare)(struct device *, void *)`
compare function used for matching component

`struct device_node *node`
of_node

`int drm_of_component_probe(struct device *dev, int (*compare_of)(struct device*, void*), const struct component_master_ops *m_ops)`

Generic probe function for a component based master

Parameters

`struct device *dev`
master device containing the OF node

`int (*compare_of)(struct device *, void *)`
compare function used for matching components

`const struct component_master_ops *m_ops`
component master ops to be used

Description

Parse the platform device OF node and bind all the components associated with the master. Interface ports are added before the encoders in order to satisfy their .bind requirements See Documentation/devicetree/bindings/graph.txt for the bindings.

Returns zero if successful, or one of the standard error codes if it fails.

```
int drm_of_find_panel_or_bridge(const struct device_node *np, int port, int endpoint,  
                               struct drm_panel **panel, struct drm_bridge **bridge)  
    return connected panel or bridge device
```

Parameters

const struct device_node *np
device tree node containing encoder output ports

int port
port in the device tree node

int endpoint
endpoint in the device tree node

struct drm_panel **panel
pointer to hold returned *drm_panel*

struct drm_bridge **bridge
pointer to hold returned *drm_bridge*

Description

Given a DT node's port and endpoint number, find the connected node and return either the associated *struct drm_panel* or *drm_bridge* device. Either **panel** or **bridge** must not be NULL.

This function is deprecated and should not be used in new drivers. Use *devm_drm_of_get_bridge()* instead.

Returns zero if successful, or one of the standard error codes if it fails.

```
int drm_of_lvds_get_dual_link_pixel_order(const struct device_node *port1, const struct  
                                          device_node *port2)  
    Get LVDS dual-link pixel order
```

Parameters

const struct device_node *port1
First DT port node of the Dual-link LVDS source

const struct device_node *port2
Second DT port node of the Dual-link LVDS source

Description

An LVDS dual-link connection is made of two links, with even pixels transitting on one link, and odd pixels on the other link. This function returns, for two ports of an LVDS dual-link source, which port shall transmit the even and odd pixels, based on the requirements of the connected sink.

The pixel order is determined from the *dual-lvds-even-pixels* and *dual-lvds-odd-pixels* properties in the sink's DT port nodes. If those properties are not present, or if their usage is not valid, this function returns -EINVAL.

If either port is not connected, this function returns -EPIPE.

port1 and **port2** are typically DT sibling nodes, but may have different parents when, for instance, two separate LVDS encoders carry the even and odd pixels.

Return

- `DRM_LVDS_DUAL_LINK_EVEN_ODD_PIXELS` - **port1** carries even pixels and **port2** carries odd pixels
- `DRM_LVDS_DUAL_LINK_ODD_EVEN_PIXELS` - **port1** carries odd pixels and **port2** carries even pixels
- `-EINVAL` - **port1** and **port2** are not connected to a dual-link LVDS sink, or the sink configuration is invalid
- `-EPIPE` - when **port1** or **port2** are not connected

int **drm_of_lvds_get_data_mapping**(const struct device_node *port)

Get LVDS data mapping

Parameters

const struct device_node *port

DT port node of the LVDS source or sink

Description

Convert DT “data-mapping” property string value into media bus format value.

Return

- `MEDIA_BUS_FMT_RGB666_1X7X3_SPWG` - data-mapping is “jeida-18”
- `MEDIA_BUS_FMT_RGB888_1X7X4_JEIDA` - data-mapping is “jeida-24”
- `MEDIA_BUS_FMT_RGB888_1X7X4_SPWG` - data-mapping is “vesa-24”
- `-EINVAL` - the “data-mapping” property is unsupported
- `-ENODEV` - the “data-mapping” property is missing

int **drm_of_get_data_lanes_count**(const struct device_node *endpoint, const unsigned int min, const unsigned int max)

Get DSI/(e)DP data lane count

Parameters

const struct device_node *endpoint

DT endpoint node of the DSI/(e)DP source or sink

const unsigned int min

minimum supported number of data lanes

const unsigned int max

maximum supported number of data lanes

Description

Count DT “data-lanes” property elements and check for validity.

Return

- min..max - positive integer count of “data-lanes” elements
- -ve - the “data-lanes” property is missing or invalid
- `-EINVAL` - the “data-lanes” property is unsupported

int **drm_of_get_data_lanes_count_ep**(const struct device_node *port, int port_reg, int reg,
const unsigned int min, const unsigned int max)

Get DSI/(e)DP data lane count by endpoint

Parameters

const struct device_node *port

DT port node of the DSI/(e)DP source or sink

int port_reg

identifier (value of reg property) of the parent port node

int reg

identifier (value of reg property) of the endpoint node

const unsigned int min

minimum supported number of data lanes

const unsigned int max

maximum supported number of data lanes

Description

Count DT “data-lanes” property elements and check for validity. This variant uses endpoint specifier.

Return

- min..max - positive integer count of “data-lanes” elements
- -EINVAL - the “data-mapping” property is unsupported
- -ENODEV - the “data-mapping” property is missing

struct *mipi_dsi_host* ***drm_of_get_dsi_bus**(struct device *dev)

find the DSI bus for a given device

Parameters

struct device *dev

parent device of display (SPI, I2C)

Description

Gets parent DSI bus for a DSI device controlled through a bus other than MIPI-DCS (SPI, I2C, etc.) using the Device Tree.

Returns pointer to *mipi_dsi_host* if successful, -EINVAL if the request is unsupported, -EPROBE_DEFER if the DSI host is found but not available, or -ENODEV otherwise.

5.27 Legacy Plane Helper Reference

This helper library contains helpers to implement primary plane support on top of the normal CRTC configuration interface. Since the legacy `drm_mode_config_funcs.set_config` interface ties the primary plane together with the CRTC state this does not allow userspace to disable the primary plane itself. The default primary plane only expose XRGB8888 and ARGB8888 as valid pixel formats for the attached framebuffer.

Drivers are highly recommended to implement proper support for primary planes, and newly merged drivers must not rely upon these transitional helpers.

The plane helpers share the function table structures with other helpers, specifically also the atomic helpers. See `struct drm_plane_helper_funcs` for the details.

```
int drm_plane_helper_update_primary(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,
                                   struct drm_modeset_acquire_ctx *ctx)
```

Helper for updating primary planes

Parameters

struct drm_plane *plane

plane to update

struct drm_crtc *crtc

the plane's new CRTC

struct drm_framebuffer *fb

the plane's new framebuffer

int crtc_x

x coordinate within CRTC

int crtc_y

y coordinate within CRTC

unsigned int crtc_w

width coordinate within CRTC

unsigned int crtc_h

height coordinate within CRTC

uint32_t src_x

x coordinate within source

uint32_t src_y

y coordinate within source

uint32_t src_w

width coordinate within source

uint32_t src_h

height coordinate within source

struct drm_modeset_acquire_ctx *ctx

modeset locking context

Description

This helper validates the given parameters and updates the primary plane.

This function is only useful for non-atomic modesetting. Don't use it in new drivers.

Return

Zero on success, or an errno code otherwise.

```
int drm_plane_helper_disable_primary(struct drm_plane *plane, struct  
                                     drm_modeset_acquire_ctx *ctx)
```

Helper for disabling primary planes

Parameters

```
struct drm_plane *plane  
    plane to disable
```

```
struct drm_modeset_acquire_ctx *ctx  
    modeset locking context
```

Description

This helper returns an error when trying to disable the primary plane.

This function is only useful for non-atomic modesetting. Don't use it in new drivers.

Return

An errno code.

```
void drm_plane_helper_destroy(struct drm_plane *plane)  
    Helper for primary plane destruction
```

Parameters

```
struct drm_plane *plane  
    plane to destroy
```

Description

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

```
int drm_plane_helper_atomic_check(struct drm_plane *plane, struct drm_atomic_state  
                                  *state)
```

Helper to check plane atomic-state

Parameters

```
struct drm_plane *plane  
    plane to check
```

```
struct drm_atomic_state *state  
    atomic state object
```

Description

Provides a default plane-state check handler for planes whose atomic-state scale and positioning are not expected to change since the plane is always a fullscreen scanout buffer.

This is often the case for the primary plane of simple framebuffers. See also [`drm_crtc_helper_atomic_check\(\)`](#) for the respective CRTC-state check helper function.

Return

Zero on success, or an errno code otherwise.

5.28 Legacy CRTC/Modeset Helper Functions Reference

The CRTC modeset helper library provides a default `set_config` implementation in [`drm_crtc_helper_set_config\(\)`](#). Plus a few other convenience functions using the same callbacks which drivers can use to e.g. restore the modeset configuration on resume with [`drm_helper_resume_force_mode\(\)`](#).

Note that this helper library doesn't track the current power state of CRTCs and encoders. It can call callbacks like [`drm_encoder_helper_funcs.dpms`](#) even though the hardware is already in the desired state. This deficiency has been fixed in the atomic helpers.

The driver callbacks are mostly compatible with the atomic modeset helpers, except for the handling of the primary plane: Atomic helpers require that the primary plane is implemented as a real standalone plane and not directly tied to the CRTC state. For easier transition this library provides functions to implement the old semantics required by the CRTC helpers using the new plane and atomic helper callbacks.

Drivers are strongly urged to convert to the atomic helpers (by way of first converting to the plane helpers). New drivers must not use these functions but need to implement the atomic interface instead, potentially using the atomic helpers for that.

These legacy modeset helpers use the same function table structures as all other modesetting helpers. See the documentation for struct [`drm_crtc_helper_funcs`](#), struct [`drm_encoder_helper_funcs`](#) and struct [`drm_connector_helper_funcs`](#).

bool **drm_helper_encoder_in_use**(struct [`drm_encoder`](#) *encoder)

check if a given encoder is in use

Parameters

struct [`drm_encoder`](#) *encoder

encoder to check

Description

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

Return

True if **encoder** is used, false otherwise.

bool **drm_helper_crtc_in_use**(struct [`drm_crtc`](#) *crtc)

check if a given CRTC is in a mode_config

Parameters

struct [`drm_crtc`](#) *crtc

CRTC to check

Description

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

Return

True if **crtc** is used, false otherwise.

void **drm_helper_disable_unused_functions**(struct *drm_device* *dev)
 disable unused objects

Parameters

struct drm_device *dev
 DRM device

Description

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

This function is part of the legacy modeset helper library and will cause major confusion with atomic drivers. This is because atomic helpers guarantee to never call `->disable()` hooks on a disabled function, or `->enable()` hooks on an enabled functions. *drm_helper_disable_unused_functions()* on the other hand throws such guarantees into the wind and calls disable hooks unconditionally on unused functions.

NOTE

bool **drm_crtc_helper_set_mode**(struct *drm_crtc* *crtc, struct *drm_display_mode* *mode, int x, int y, struct *drm_framebuffer* *old_fb)
 internal helper to set a mode

Parameters

struct drm_crtc *crtc
 CRTC to program

struct drm_display_mode *mode
 mode to use

int x
 horizontal offset into the surface

int y
 vertical offset into the surface

struct drm_framebuffer *old_fb
 old framebuffer, for cleanup

Description

Try to set **mode** on **crtc**. Give **crtc** and its associated connectors a chance to fixup or reject the mode prior to trying to set it. This is an internal helper that drivers could e.g. use to update properties that require the entire output pipe to be disabled and re-enabled in a new configuration. For example for changing whether audio is enabled on a hdmi link or for changing

panel fitter or dither attributes. It is also called by the `drm_crtc_helper_set_config()` helper function to drive the mode setting sequence.

Return

True if the mode was set successfully, false otherwise.

int **drm_crtc_helper_atomic_check**(struct *drm_crtc* *crtc, struct *drm_atomic_state* *state)
Helper to check CRTC atomic-state

Parameters

struct drm_crtc *crtc
CRTC to check

struct drm_atomic_state *state
atomic state object

Description

Provides a default CRTC-state check handler for CRTCs that only have one primary plane attached to it.

This is often the case for the CRTC of simple framebuffers. See also `drm_plane_helper_atomic_check()` for the respective plane-state check helper function.

Return

Zero on success, or an errno code otherwise.

int **drm_crtc_helper_set_config**(struct *drm_mode_set* *set, struct *drm_modeset_acquire_ctx* *ctx)
set a new config from userspace

Parameters

struct drm_mode_set *set
mode set configuration

struct drm_modeset_acquire_ctx *ctx
lock acquire context, not used here

Description

The `drm_crtc_helper_set_config()` helper function implements the of `drm_crtc_funcs.set_config` callback for drivers using the legacy CRTC helpers.

It first tries to locate the best encoder for each connector by calling the connector **drm_connector_helper_funcs.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 `drm_crtc_helper_funcs.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. Alternatively it can also use the dpms and disable helper operations. For details see [struct `drm_crtc_helper_funcs`](#) and [struct `drm_encoder_helper_funcs`](#).

This function is deprecated. New drivers must implement atomic modeset support, for which this function is unsuitable. Instead drivers should use [`drm_atomic_helper_set_config\(\)`](#).

Return

Returns 0 on success, negative errno numbers on failure.

`int` **drm_helper_connector_dpms**(struct [`drm_connector`](#) *connector, int mode)
connector dpms helper implementation

Parameters

struct `drm_connector` *connector
affected connector

int mode
DPMS mode

Description

The [`drm_helper_connector_dpms\(\)`](#) helper function implements the [`drm_connector_funcs.dpms`](#) callback for drivers using the legacy CRTC helpers.

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 [`drm_crtc_helper_funcs.dpms`](#) and [`drm_encoder_helper_funcs.dpms`](#) callbacks provided by the driver.

This function is deprecated. New drivers must implement atomic modeset support, where DPMS is handled in the DRM core.

Return

Always returns 0.

`void` **drm_helper_resume_force_mode**(struct [`drm_device`](#) *dev)
force-restore mode setting configuration

Parameters

struct `drm_device` *dev
drm_device which should be restored

Description

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

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

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

This function is deprecated. New drivers should implement atomic mode- setting and use the atomic suspend/resume helpers.

See also: [`drm_atomic_helper_suspend\(\)`](#), [`drm_atomic_helper_resume\(\)`](#)

int **drm_helper_force_disable_all**(struct [`drm_device`](#) *dev)

Forcibly turn off all enabled CRTCs

Parameters

struct [`drm_device`](#) *dev

DRM device whose CRTCs to turn off

Description

Drivers may want to call this on unload to ensure that all displays are unlit and the GPU is in a consistent, low power state. Takes modeset locks.

Note

This should only be used by non-atomic legacy drivers. For an atomic version look at [`drm_atomic_helper_shutdown\(\)`](#).

Return

Zero on success, error code on failure.

5.29 Privacy-screen class

This class allows non KMS drivers, from e.g. `drivers/platform/x86` to register a privacy-screen device, which the KMS drivers can then use to implement the standard privacy-screen properties, see [*Standard Connector Properties*](#).

KMS drivers using a privacy-screen class device are advised to use the [`drm_connector_attach_privacy_screen_provider\(\)`](#) and [`drm_connector_update_privacy_screen\(\)`](#) helpers for dealing with this.

struct **drm_privacy_screen_ops**

drm_privacy_screen operations

Definition:

```
struct drm_privacy_screen_ops {
    int (*set_sw_state)(struct drm_privacy_screen *priv, enum drm_privacy_
→screen_status sw_state);
    void (*get_hw_state)(struct drm_privacy_screen *priv);
};
```

Members

set_sw_state

Called to request a change of the privacy-screen state. The privacy-screen class code contains a check to avoid this getting called when the `hw_state` reports the state is locked.

It is the driver's responsibility to update `sw_state` and `hw_state`. This is always called with the `drm_privacy_screen`'s lock held.

get_hw_state

Called to request that the driver gets the current privacy-screen state from the hardware and then updates `sw_state` and `hw_state` accordingly. This will be called by the core just before the privacy-screen is registered in sysfs.

Description

Defines the operations which the privacy-screen class code may call. These functions should be implemented by the privacy-screen driver.

struct **drm_privacy_screen**

central privacy-screen structure

Definition:

```
struct drm_privacy_screen {
    struct device dev;
    struct mutex lock;
    struct list_head list;
    struct blocking_notifier_head notifier_head;
    const struct drm_privacy_screen_ops *ops;
    enum drm_privacy_screen_status sw_state;
    enum drm_privacy_screen_status hw_state;
    void *drvdata;
};
```

Members

dev

device used to register the privacy-screen in sysfs.

lock

mutex protection all fields in this struct.

list

privacy-screen devices list list-entry.

notifier_head

privacy-screen notifier head.

ops

struct drm_privacy_screen_ops for this privacy-screen. This is NULL if the driver has unregistered the privacy-screen.

sw_state

The privacy-screen's software state, see *Standard Connector Properties* for more info.

hw_state

The privacy-screen's hardware state, see *Standard Connector Properties* for more info.

drvdata

Private data owned by the privacy screen provider

Description

Central privacy-screen structure, this contains the struct device used to register the screen in sysfs, the screen's state, ops, etc.

struct **drm_privacy_screen_lookup**
static privacy-screen lookup list entry

Definition:

```
struct drm_privacy_screen_lookup {
    struct list_head list;
    const char *dev_id;
    const char *con_id;
    const char *provider;
};
```

Members

list

Lookup list list-entry.

dev_id

Consumer device name or NULL to match all devices.

con_id

Consumer connector name or NULL to match all connectors.

provider

dev_name() of the privacy_screen provider.

Description

Used for the static lookup-list for mapping privacy-screen consumer dev-connector pairs to a privacy-screen provider.

void **drm_privacy_screen_lookup_add**(struct *drm_privacy_screen_lookup* *lookup)
add an entry to the static privacy-screen lookup list

Parameters

struct **drm_privacy_screen_lookup** *lookup
lookup list entry to add

Description

Add an entry to the static privacy-screen lookup list. Note the struct `list_head` which is part of the *struct drm_privacy_screen_lookup* gets added to a list owned by the privacy-screen core. So the passed in *struct drm_privacy_screen_lookup* must not be free-ed until it is removed from the lookup list by calling *drm_privacy_screen_lookup_remove()*.

void **drm_privacy_screen_lookup_remove**(struct *drm_privacy_screen_lookup* *lookup)
remove an entry to the static privacy-screen lookup list

Parameters

struct **drm_privacy_screen_lookup** *lookup
lookup list entry to remove

Description

Remove an entry previously added with `drm_privacy_screen_lookup_add()` from the static privacy-screen lookup list.

`struct drm_privacy_screen *drm_privacy_screen_get(struct device *dev, const char *con_id)`
get a privacy-screen provider

Parameters

struct device *dev
consumer-device for which to get a privacy-screen provider

const char *con_id
(video)connector name for which to get a privacy-screen provider

Description

Get a privacy-screen provider for a privacy-screen attached to the display described by the **dev** and **con_id** parameters.

Return

- A pointer to a `struct drm_privacy_screen` on success.
- `ERR_PTR(-ENODEV)` if no matching privacy-screen is found
- **`ERR_PTR(-EPROBE_DEFER)` if there is a matching privacy-screen,**
but it has not been registered yet.

`void drm_privacy_screen_put(struct drm_privacy_screen *priv)`
release a privacy-screen reference

Parameters

struct *drm_privacy_screen* *priv
privacy screen reference to release

Description

Release a privacy-screen provider reference gotten through `drm_privacy_screen_get()`. May be called with a NULL or `ERR_PTR`, in which case it is a no-op.

`int drm_privacy_screen_set_sw_state(struct drm_privacy_screen *priv, enum drm_privacy_screen_status sw_state)`
set a privacy-screen's sw-state

Parameters

struct *drm_privacy_screen* *priv
privacy screen to set the sw-state for

enum *drm_privacy_screen_status* sw_state
new sw-state value to set

Description

Set the sw-state of a privacy screen. If the privacy-screen is not in a locked hw-state, then the actual and hw-state of the privacy-screen will be immediately updated to the new value. If the privacy-screen is in a locked hw-state, then the new sw-state will be remembered as the requested state to put the privacy-screen in when it becomes unlocked.

Return

0 on success, negative error code on failure.

```
void drm_privacy_screen_get_state(struct drm_privacy_screen *priv, enum  
                                drm_privacy_screen_status *sw_state_ret, enum  
                                drm_privacy_screen_status *hw_state_ret)
```

get privacy-screen's current state

Parameters

```
struct drm_privacy_screen *priv  
    privacy screen to get the state for
```

```
enum drm_privacy_screen_status *sw_state_ret  
    address where to store the privacy-screens current sw-state
```

```
enum drm_privacy_screen_status *hw_state_ret  
    address where to store the privacy-screens current hw-state
```

Description

Get the current state of a privacy-screen, both the sw-state and the hw-state.

```
int drm_privacy_screen_register_notifier(struct drm_privacy_screen *priv, struct  
                                         notifier_block *nb)
```

register a notifier

Parameters

```
struct drm_privacy_screen *priv  
    Privacy screen to register the notifier with
```

```
struct notifier_block *nb  
    Notifier-block for the notifier to register
```

Description

Register a notifier with the privacy-screen to be notified of changes made to the privacy-screen state from outside of the privacy-screen class. E.g. the state may be changed by the hardware itself in response to a hotkey press.

The notifier is called with no locks held. The new hw_state and sw_state can be retrieved using the *drm_privacy_screen_get_state()* function. A pointer to the *drm_privacy_screen*'s struct is passed as the void *data argument of the notifier_block's notifier_call.

The notifier will NOT be called when changes are made through *drm_privacy_screen_set_sw_state()*. It is only called for external changes.

Return

0 on success, negative error code on failure.

```
int drm_privacy_screen_unregister_notifier(struct drm_privacy_screen *priv, struct  
                                           notifier_block *nb)
```

unregister a notifier

Parameters

```
struct drm_privacy_screen *priv  
    Privacy screen to register the notifier with
```

struct notifier_block *nb

Notifier-block for the notifier to register

Description

Unregister a notifier registered with *drm_privacy_screen_register_notifier()*.

Return

0 on success, negative error code on failure.

struct *drm_privacy_screen* ***drm_privacy_screen_register**(struct device *parent, const struct *drm_privacy_screen_ops* *ops, void *data)

register a privacy-screen

Parameters

struct device *parent

parent-device for the privacy-screen

const struct drm_privacy_screen_ops *ops

struct drm_privacy_screen_ops pointer with ops for the privacy-screen

void *data

Private data owned by the privacy screen provider

Description

Create and register a privacy-screen.

Return

- A pointer to the created privacy-screen on success.
- An ERR_PTR(errno) on failure.

void **drm_privacy_screen_unregister**(struct *drm_privacy_screen* *priv)

unregister privacy-screen

Parameters

struct drm_privacy_screen *priv

privacy-screen to unregister

Description

Unregister a privacy-screen registered with *drm_privacy_screen_register()*. May be called with a NULL or ERR_PTR, in which case it is a no-op.

void **drm_privacy_screen_call_notifier_chain**(struct *drm_privacy_screen* *priv)

notify consumers of state change

Parameters

struct drm_privacy_screen *priv

Privacy screen to register the notifier with

Description

A privacy-screen provider driver can call this functions upon external changes to the privacy-screen state. E.g. the state may be changed by the hardware itself in response to a hotkey press. This function must be called without holding the privacy-screen lock. the

driver must update `sw_state` and `hw_state` to reflect the new state before calling this function. The expected behavior from the driver upon receiving an external state change event is: 1. Take the lock; 2. Update `sw_state` and `hw_state`; 3. Release the lock. 4. Call *`drm_privacy_screen_call_notifier_chain()`*.

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.

6.1 libdrm Device Lookup

BEWARE THE DRAGONS! MIND THE TRAPDOORS!

In an attempt to warn anyone else who's trying to figure out what's going on here, I'll try to summarize the story. First things first, let's clear up the names, because the kernel internals, libdrm and the ioctls are all named differently:

- GET_UNIQUE ioctl, implemented by `drm_getunique` is wrapped up in libdrm through the `drmGetBusid` function.
- The libdrm `drmSetBusid` function is backed by the SET_UNIQUE ioctl. All that code is nerved in the kernel with `drm_invalid_op()`.
- The internal `set_busid` kernel functions and driver callbacks are exclusively use by the SET_VERSION ioctl, because only drm 1.0 (which is nerved) allowed userspace to set the busid through the above ioctl.
- Other ioctls and functions involved are named consistently.

For anyone wondering what's the difference between drm 1.1 and 1.4: Correctly handling pci domains in the busid on ppc. Doing this correctly was only implemented in libdrm in 2010, hence can't be nerved yet. No one knows what's special with drm 1.2 and 1.3.

Now the actual horror story of how device lookup in drm works. At large, there's 2 different ways, either by busid, or by device driver name.

Opening by busid is fairly simple:

1. First call SET_VERSION to make sure pci domains are handled properly. As a side-effect this fills out the unique name in the master structure.

2. Call `GET_UNIQUE` to read out the unique name from the master structure, which matches the busid thanks to step 1. If it doesn't, proceed to try the next device node.

Opening by name is slightly different:

1. Directly call `VERSION` to get the version and to match against the driver name returned by that ioctl. Note that `SET_VERSION` is not called, which means the unique name for the master node just opening is `_not_` filled out. This despite that with current drm device nodes are always bound to one device, and can't be runtime assigned like with drm 1.0.
2. Match driver name. If it mismatches, proceed to the next device node.
3. Call `GET_UNIQUE`, and check whether the unique name has length zero (by checking that the first byte in the string is 0). If that's not the case libdrm skips and proceeds to the next device node. Probably this is just copy-pasta from drm 1.0 times where a set unique name meant that the driver was in use already, but that's just conjecture.

Long story short: To keep the open by name logic working, `GET_UNIQUE` must `_not_` return a unique string when `SET_VERSION` hasn't been called yet, otherwise libdrm breaks. Even when that unique string can't ever change, and is totally irrelevant for actually opening the device because runtime assignable device instances were only support in drm 1.0, which is long dead. But the libdrm code in `drmOpenByName` somehow survived, hence this can't be broken.

6.2 Primary Nodes, DRM Master and Authentication

`struct drm_master` is used to track groups of clients with open primary/legacy device nodes. For every `struct drm_file` which has had at least once successfully became the device master (either through the `SET_MASTER` IOCTL, or implicitly through opening the primary device node when no one else is the current master that time) there exists one `drm_master`. This is noted in `drm_file.is_master`. All other clients have just a pointer to the `drm_master` they are associated with.

In addition only one `drm_master` can be the current master for a `drm_device`. It can be switched through the `DROP_MASTER` and `SET_MASTER` IOCTL, or implicitly through closing/opening the primary device node. See also `drm_is_current_master()`.

Clients can authenticate against the current master (if it matches their own) using the `GET_MAGIC` and `AUTHMAGIC` IOCTLs. Together with exchanging masters, this allows controlled access to the device for an entire group of mutually trusted clients.

`bool drm_is_current_master(struct drm_file *fpriv)`
checks whether **fpriv** is the current master

Parameters

`struct drm_file *fpriv`
DRM file private

Description

Checks whether **fpriv** is current master on its device. This decides whether a client is allowed to run `DRM_MASTER` IOCTLs.

Most of the modern IOCTL which require `DRM_MASTER` are for kernel modesetting - the current master is assumed to own the non-shareable display hardware.

struct *drm_master* ***drm_master_get**(struct *drm_master* *master)
 reference a master pointer

Parameters

struct *drm_master* *master
 struct *drm_master*

Description

Increments the reference count of **master** and returns a pointer to **master**.

struct *drm_master* ***drm_file_get_master**(struct *drm_file* *file_priv)
 reference *drm_file.master* of **file_priv**

Parameters

struct *drm_file* *file_priv
 DRM file private

Description

Increments the reference count of **file_priv**'s *drm_file.master* and returns the *drm_file.master*. If **file_priv** has no *drm_file.master*, returns NULL.

Master pointers returned from this function should be unreferenced using *drm_master_put()*.

void **drm_master_put**(struct *drm_master* **master)
 unreference and clear a master pointer

Parameters

struct *drm_master* **master
 pointer to a pointer of struct *drm_master*

Description

This decrements the *drm_master* behind **master** and sets it to NULL.

struct **drm_master**
 drm master structure

Definition:

```
struct drm_master {
    struct kref refcount;
    struct drm_device *dev;
    char *unique;
    int unique_len;
    struct idr magic_map;
    void *driver_priv;
    struct drm_master *lessor;
    int lessee_id;
    struct list_head lessee_list;
    struct list_head lessees;
    struct idr leases;
    struct idr lessee_idr;
};
```

Members

refcount

Refcount for this master object.

dev

Link back to the DRM device

unique

Unique identifier: e.g. busid. Protected by *drm_device.master_mutex*.

unique_len

Length of unique field. Protected by *drm_device.master_mutex*.

magic_map

Map of used authentication tokens. Protected by *drm_device.master_mutex*.

driver_priv

Pointer to driver-private information.

lessor

Lease grantor, only set if this *struct drm_master* represents a lessee holding a lease of objects from **lessor**. Full owners of the device have this set to NULL.

The lessor does not change once it's set in *drm_lease_create()*, and each lessee holds a reference to its lessor that it releases upon being destroyed in *drm_lease_destroy()*.

See also the *section on display resource leasing*.

lessee_id

ID for lessees. Owners (i.e. **lessor** is NULL) always have ID 0. Protected by *drm_device.mode_config's drm_mode_config.idr_mutex*.

lessee_list

List entry of lessees of **lessor**, where they are linked to **lessees**. Not used for owners. Protected by *drm_device.mode_config's drm_mode_config.idr_mutex*.

lessees

List of *drm_masters* leasing from this one. Protected by *drm_device.mode_config's drm_mode_config.idr_mutex*.

This list is empty if no leases have been granted, or if all lessees have been destroyed. Since lessors are referenced by all their lessees, this master cannot be destroyed unless the list is empty.

leases

Objects leased to this *drm_master*. Protected by *drm_device.mode_config's drm_mode_config.idr_mutex*.

Objects are leased all together in *drm_lease_create()*, and are removed all together when the lease is revoked.

lessee_idr

All lessees under this owner (only used where **lessor** is NULL). Protected by *drm_device.mode_config's drm_mode_config.idr_mutex*.

Description

Note that master structures are only relevant for the legacy/primary device nodes, hence there can only be one per device, not one per *drm_minor*.

6.3 DRM Display Resource Leasing

DRM leases provide information about whether a DRM master may control a DRM mode setting object. This enables the creation of multiple DRM masters that manage subsets of display resources.

The original DRM master of a device ‘owns’ the available drm resources. It may create additional DRM masters and ‘lease’ resources which it controls to the new DRM master. This gives the new DRM master control over the leased resources until the owner revokes the lease, or the new DRM master is closed. Some helpful terminology:

- An ‘owner’ is a `struct drm_master` that is not leasing objects from another `struct drm_master`, and hence ‘owns’ the objects. The owner can be identified as the `struct drm_master` for which `drm_master.lessor` is NULL.
- A ‘lessor’ is a `struct drm_master` which is leasing objects to one or more other `struct drm_master`. Currently, lessees are not allowed to create sub-leases, hence the lessor is the same as the owner.
- A ‘lessee’ is a `struct drm_master` which is leasing objects from some other `struct drm_master`. Each lessee only leases resources from a single lessor recorded in `drm_master.lessor`, and holds the set of objects that it is leasing in `drm_master.leases`.
- A ‘lease’ is a contract between the lessor and lessee that identifies which resources may be controlled by the lessee. All of the resources that are leased must be owned by or leased to the lessor, and lessors are not permitted to lease the same object to multiple lessees.

The set of objects any `struct drm_master` ‘controls’ is limited to the set of objects it leases (for lessees) or all objects (for owners).

Objects not controlled by a `struct drm_master` cannot be modified through the various state manipulating ioctls, and any state reported back to user space will be edited to make them appear idle and/or unusable. For instance, connectors always report ‘disconnected’, while encoders report no possible crtcs or clones.

Since each lessee may lease objects from a single lessor, display resource leases form a tree of `struct drm_master`. As lessees are currently not allowed to create sub-leases, the tree depth is limited to 1. All of these get activated simultaneously when the top level device owner changes through the SETMASTER or DROPMASTER IOCTL, so `drm_device.master` points to the owner at the top of the lease tree (i.e. the `struct drm_master` for which `drm_master.lessor` is NULL). The full list of lessees that are leasing objects from the owner can be searched via the owner’s `drm_master.lessee_idr`.

6.4 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.

6.5 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. For a complete list of driver-independent ioctls that can be used on render nodes, see the ioctls marked `DRM_RENDER_ALLOW` in `drm_ioctl.c`. 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.

6.6 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.

6.6.1 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.

6.6.2 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`.

6.6.3 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.

6.7 IOCTL Support on Device Nodes

First things first, driver private IOCTLs should only be needed for drivers supporting rendering. Kernel modesetting is all standardized, and extended through properties. There are a few exceptions in some existing drivers, which define IOCTL for use by the display DRM master, but they all predate properties.

Now if you do have a render driver you always have to support it through driver private properties. There's a few steps needed to wire all the things up.

First you need to define the structure for your IOCTL in your driver private UAPI header in `include/uapi/drm/my_driver_drm.h`:

```
struct my_driver_operation {
    u32 some_thing;
    u32 another_thing;
};
```


Please make sure that you follow all the best practices from `Documentation/process/botching-up-ioctls.rst`. Note that `drm_ioctl()` automatically zero-extends structures, hence make sure you can add more stuff at the end, i.e. don't put a variable sized array there.

Then you need to define your IOCTL number, using one of `DRM_IOCTL()`, `DRM_IOR()`, `DRM_IOW()` or `DRM_IOWR()`. It must start with the `DRM_IOCTL_` prefix:

```
#define DRM_IOCTL_MY_DRIVER_OPERATION *          DRM_IOW(DRM_COMMAND_BASE, u  
↪ struct my_driver_operation)
```

DRM driver private IOCTL must be in the range from `DRM_COMMAND_BASE` to `DRM_COMMAND_END`. Finally you need an array of `struct drm_ioctl_desc` to wire up the handlers and set the access rights:

```
static const struct drm_ioctl_desc my_driver_ioctls[] = {  
    DRM_IOCTL_DEF_DRV(MY_DRIVER_OPERATION, my_driver_operation,  
        DRM_AUTH|DRM_RENDER_ALLOW),  
};
```

And then assign this to the `drm_driver.ioctls` field in your driver structure.

See the separate chapter on *file operations* for how the driver-specific IOCTLs are wired up.

6.7.1 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 unprivileged 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.

drm_ioctl_t

Typedef: DRM ioctl function type.

Syntax

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

Parameters

struct drm_device *dev

DRM device inode

void *data

private pointer of the ioctl call

struct drm_file *file_priv

DRM file this ioctl was made on

Description

This is the DRM ioctl typedef. Note that `drm_ioctl()` has already copied **data** into kernel-space, and will also copy it back, depending upon the read/write settings in the ioctl command code.

drm_ioctl_compat_t

Typedef: compatibility DRM ioctl function type.

Syntax

```
typedef int drm_ioctl_compat_t (struct file *filp, unsigned int cmd,
unsigned long arg)
```

Parameters

struct file *filp

file pointer

unsigned int cmd

ioctl command code

unsigned long arg

DRM file this ioctl was made on

Description

Just a typedef to make declaring an array of compatibility handlers easier. New drivers shouldn't screw up the structure layout for their ioctl structures and hence never need this.

enum **drm_ioctl_flags**

DRM ioctl flags

Constants

DRM_AUTH

This is for ioctl which are used for rendering, and require that the file descriptor is either for a render node, or if it's a legacy/primary node, then it must be authenticated.

DRM_MASTER

This must be set for any ioctl which can change the modeset or display state. Userspace must call the ioctl through a primary node, while it is the active master.

Note that read-only modeset ioctl can also be called by unauthenticated clients, or when a master is not the currently active one.

DRM_ROOT_ONLY

Anything that could potentially wreck a master file descriptor needs to have this flag set. Current that's only for the SETMASTER and DROPMASTER ioctl, which e.g. logind can call to force a non-behaving master (display compositor) into compliance.

This is equivalent to callers with the SYSADMIN capability.

DRM_UNLOCKED

Whether *drm_ioctl_desc.func* should be called with the DRM BKL held or not. Enforced as the default for all modern drivers, hence there should never be a need to set this flag.

Do not use anywhere else than for the VBLANK_WAIT IOCTL, which is the only legacy IOCTL which needs this.

DRM_RENDER_ALLOW

This is used for all ioctl needed for rendering only, for drivers which support render nodes. This should be all new render drivers, and hence it should be always set for any ioctl with DRM_AUTH set. Note though that read-only query ioctl might have this set, but have not set DRM_AUTH because they do not require authentication.

Description

Various flags that can be set in *drm_ioctl_desc.flags* to control how userspace can use a given ioctl.

struct **drm_ioctl_desc**

DRM driver ioctl entry

Definition:

```
struct drm_ioctl_desc {
    unsigned int cmd;
    enum drm_ioctl_flags flags;
    drm_ioctl_t *func;
    const char *name;
};
```

Members

cmd
ioctl command number, without flags

flags
a bitmask of *enum drm_ioctl_flags*

func
handler for this ioctl

name
user-readable name for debug output

Description

For convenience it's easier to create these using the *DRM_IOCTL_DEF_DRV()* macro.

DRM_IOCTL_DEF_DRV

DRM_IOCTL_DEF_DRV (ioctl, _func, _flags)
helper macro to fill out a *struct drm_ioctl_desc*

Parameters

ioctl
ioctl command suffix

_func
handler for the ioctl

_flags
a bitmask of *enum drm_ioctl_flags*

Description

Small helper macro to create a *struct drm_ioctl_desc* entry. The ioctl command number is constructed by prepending *DRM_IOCTL_* and passing that to *DRM_IOCTL_NR()*.

int **drm_noop**(struct *drm_device* *dev, void *data, struct *drm_file* *file_priv)
DRM no-op ioctl implementation

Parameters

struct drm_device *dev
DRM device for the ioctl

void *data
data pointer for the ioctl

struct drm_file *file_priv
DRM file for the ioctl call

Description

This no-op implementation for drm ioctls is useful for deprecated functionality where we can't return a failure code because existing userspace checks the result of the ioctl, but doesn't care about the action.

Always returns successfully with 0.

int **drm_invalid_op**(struct *drm_device* *dev, void *data, struct *drm_file* *file_priv)

DRM invalid ioctl implementation

Parameters

struct drm_device *dev

DRM device for the ioctl

void *data

data pointer for the ioctl

struct drm_file *file_priv

DRM file for the ioctl call

Description

This no-op implementation for drm ioctls is useful for deprecated functionality where we really don't want to allow userspace to call the ioctl any more. This is the case for old ums interfaces for drivers that transitioned to kms gradually and so kept the old legacy tables around. This only applies to radeon and i915 kms drivers, other drivers shouldn't need to use this function.

Always fails with a return value of -EINVAL.

long **drm_ioctl**(struct file *filp, unsigned int cmd, unsigned long arg)

ioctl callback implementation for DRM drivers

Parameters

struct file *filp

file this ioctl is called on

unsigned int cmd

ioctl cmd number

unsigned long arg

user argument

Description

Looks up the ioctl function in the DRM core and the driver dispatch table, stored in *drm_driver.ioctls*. It checks for necessary permission by calling *drm_ioctl_permit()*, and dispatches to the respective function.

Return

Zero on success, negative error code on failure.

bool **drm_ioctl_flags**(unsigned int nr, unsigned int *flags)

Check for core ioctl and return ioctl permission flags

Parameters

unsigned int nr

ioctl number

unsigned int *flags

where to return the ioctl permission flags

Description

This ioctl is only used by the vmwgfx driver to augment the access checks done by the drm core and insofar a pretty decent layering violation. This shouldn't be used by any drivers.

Return

True if the **nr** corresponds to a DRM core ioctl number, false otherwise.

long **drm_compat_ioctl**(struct file *filp, unsigned int cmd, unsigned long arg)

32bit IOCTL compatibility handler for DRM drivers

Parameters

struct file *filp

file this ioctl is called on

unsigned int cmd

ioctl cmd number

unsigned long arg

user argument

Description

Compatibility handler for 32 bit userspace running on 64 kernels. All actual IOCTL handling is forwarded to `drm_ioctl()`, while marshalling structures as appropriate. Note that this only handles DRM core IOCTLs, if the driver has botched IOCTL itself, it must handle those by wrapping this function.

Return

Zero on success, negative error code on failure.

6.8 Testing and validation

6.8.1 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 test-cases in IGT for that feature, if such a test can be reasonably made using IGT for the target hardware.

6.8.2 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/>.

6.8.3 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.

Following is an example of using a VM with a shared directory with the host machine to run igt-tests. This example uses virtme:

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

Run the igt-tests in the guest machine. This example runs the 'kms_flip' tests:

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

In this example, instead of building the igt_runner, Piglit is used (-p option). It creates an HTML summary of the test results and saves them in the folder "igt-gpu-tools/results". It executes only the igt-tests matching the -t option.

6.8.4 Display CRC Support

DRM device drivers can provide to userspace CRC information of each frame as it reached a given hardware component (a CRC sampling "source").

Userspace can control generation of CRCs in a given CRTC by writing to the file `dri/0/crtc-N/crc/control` in debugfs, with N being the *index of the CRTC*. Accepted values are source names (which are driver-specific) and the "auto" keyword, which will let the driver select a default source of frame CRCs for this CRTC.

Once frame CRC generation is enabled, userspace can capture them by reading the `dri/0/crtc-N/crc/data` file. Each line in that file contains the frame number in the first field and then a number of unsigned integer fields containing the CRC data. Fields are separated by a single space and the number of CRC fields is source-specific.

Note that though in some cases the CRC is computed in a specified way and on the frame contents as supplied by userspace (eDP 1.3), in general the CRC computation is performed in an unspecified way and on frame contents that have been already processed in also an unspecified way and thus userspace cannot rely on being able to generate matching CRC values for the frame contents that it submits. In this general case, the maximum userspace can do is to compare the reported CRCs of frames that should have the same contents.

On the driver side the implementation effort is minimal, drivers only need to implement `drm_crtc_funcs.set_crc_source` and `drm_crtc_funcs.verify_crc_source`. The debugfs files are automatically set up if those vfuncs are set. CRC samples need to be captured in the driver by calling `drm_crtc_add_crc_entry()`. Depending on the driver and HW requirements, `drm_crtc_funcs.set_crc_source` may result in a commit (even a full modeset).

CRC results must be reliable across non-full-modeset atomic commits, so if a commit via `DRM_IOCTL_MODE_ATOMIC` would disable or otherwise interfere with CRC generation, then the driver must mark that commit as a full modeset (`drm_atomic_crtc_needs_modeset()` should return true). As a result, to ensure consistent results, generic userspace must re-setup CRC generation after a legacy `SETCRTC` or an atomic commit with `DRM_MODE_ATOMIC_ALLOW_MODESET`.

```
int drm_crtc_add_crc_entry(struct drm_crtc *crtc, bool has_frame, uint32_t frame, uint32_t
                        *crcs)
```

Add entry with CRC information for a frame

Parameters

struct drm_crtc *crtc

CRTC to which the frame belongs

bool has_frame

whether this entry has a frame number to go with

uint32_t frame

number of the frame these CRCs are about

uint32_t *crcs

array of CRC values, with length matching `#drm_crtc_crc.values_cnt`

Description

For each frame, the driver polls the source of CRCs for new data and calls this function to add them to the buffer from where userspace reads.

6.8.5 Debugfs Support

DRM_DEBUGFS_GPUVA_INFO

`DRM_DEBUGFS_GPUVA_INFO` (show, data)

`drm_info_list` entry to dump a GPU VA space

Parameters

show

the `drm_info_list`'s show callback

data

driver private data

Description

Drivers should use this macro to define a `drm_info_list` entry to provide a debugfs file for dumping the GPU VA space regions and mappings.

For each DRM GPU VA space drivers should call `drm_debugfs_gpuva_info()` from their **show** callback.

struct **drm_info_list**

debugfs info list entry

Definition:

```
struct drm_info_list {
    const char *name;
    int (*show)(struct seq_file*, void*);
    u32 driver_features;
    void *data;
};
```

Members

name

file name

show

Show callback. `seq_file->private` will be set to the *struct drm_info_node* corresponding to the instance of this info on a given *struct drm_minor*.

driver_features

Required driver features for this entry

data

Driver-private data, should not be device-specific.

Description

This structure represents a debugfs file to be created by the drm core.

struct **drm_info_node**

Per-minor debugfs node structure

Definition:

```
struct drm_info_node {
    struct drm_minor *minor;
    const struct drm_info_list *info_ent;
};
```

Members

minor

struct drm_minor for this node.

info_ent

template for this node.

Description

This structure represents a debugfs file, as an instantiation of a *struct drm_info_list* on a *struct drm_minor*.

FIXME:

No it doesn't make a hole lot of sense that we duplicate debugfs entries for both the render and the primary nodes, but that's how this has organically grown. It should probably be fixed, with a compatibility link, if needed.

struct **drm_debugfs_info**
debugfs info list entry

Definition:

```
struct drm_debugfs_info {  
    const char *name;  
    int (*show)(struct seq_file*, void*);  
    u32 driver_features;  
    void *data;  
};
```

Members

name
File name

show
Show callback. `seq_file->private` will be set to the *struct drm_debugfs_entry* corresponding to the instance of this info on a given *struct drm_device*.

driver_features
Required driver features for this entry.

data
Driver-private data, should not be device-specific.

Description

This structure represents a debugfs file to be created by the drm core.

struct **drm_debugfs_entry**
Per-device debugfs node structure

Definition:

```
struct drm_debugfs_entry {  
    struct drm_device *dev;  
    struct drm_debugfs_info file;  
    struct list_head list;  
};
```

Members

dev
struct drm_device for this node.

file
Template for this node.

list
Linked list of all device nodes.

Description

This structure represents a debugfs file, as an instantiation of a *struct drm_debugfs_info* on a *struct drm_device*.

int **drm_debugfs_gpuva_info**(struct seq_file *m, struct *drm_gpuva_manager* *mgr)
dump the given DRM GPU VA space

Parameters

struct seq_file *m
pointer to the seq_file to write

struct drm_gpuva_manager *mgr
the *drm_gpuva_manager* representing the GPU VA space

Description

Dumps the GPU VA mappings of a given DRM GPU VA manager.

For each DRM GPU VA space drivers should call this function from their *drm_info_list*'s show callback.

Return

0 on success, -ENODEV if the mgr is not initialized

void **drm_debugfs_create_files**(const struct *drm_info_list* *files, int count, struct dentry *root, struct *drm_minor* *minor)

Initialize a given set of debugfs files for DRM minor

Parameters

const struct drm_info_list *files
The array of files to create

int count
The number of files given

struct dentry *root
DRI debugfs dir entry.

struct drm_minor *minor
device minor number

Description

Create a given set of debugfs files represented by an array of *struct drm_info_list* in the given root directory. These files will be removed automatically on *drm_debugfs_cleanup*().

void **drm_debugfs_add_file**(struct *drm_device* *dev, const char *name, int (*show)(struct seq_file*, void*), void *data)

Add a given file to the DRM device debugfs file list

Parameters

struct drm_device *dev
drm device for the ioctl

const char *name
debugfs file name

int (*show)(struct seq_file*, void*)
show callback

void *data
driver-private data, should not be device-specific

Description

Add a given file entry to the DRM device debugfs file list to be created on `drm_debugfs_init`.

```
void drm_debugfs_add_files(struct drm_device *dev, const struct drm_debugfs_info *files,  
                           int count)
```

Add an array of files to the DRM device debugfs file list

Parameters

struct *drm_device* *dev
drm device for the ioctl

const struct *drm_debugfs_info* *files
The array of files to create

int count
The number of files given

Description

Add a given set of debugfs files represented by an array of *struct *drm_debugfs_info** in the DRM device debugfs file list.

6.9 Sysfs Support

DRM provides very little additional support to drivers for sysfs interactions, beyond just all the standard stuff. Drivers who want to expose additional sysfs properties and property groups can attach them at either *drm_device.dev* or *drm_connector.kdev*.

Registration is automatically handled when calling *drm_dev_register()*, or *drm_connector_register()* in case of hot-plugged connectors. Unregistration is also automatically handled by *drm_dev_unregister()* and *drm_connector_unregister()*.

```
void drm_sysfs_hotplug_event(struct drm_device *dev)  
    generate a DRM uevent
```

Parameters

struct *drm_device* *dev
DRM device

Description

Send a uevent for the DRM device specified by **dev**. Currently we only set HOTPLUG=1 in the uevent environment, but this could be expanded to deal with other types of events.

Any new uapi should be using the *drm_sysfs_connector_status_event()* for uevents on connector status change.

```
void drm_sysfs_connector_hotplug_event(struct drm_connector *connector)  
    generate a DRM uevent for any connector change
```

Parameters

struct *drm_connector* *connector
connector which has changed

Description

Send a uevent for the DRM connector specified by **connector**. This will send a uevent with the properties HOTPLUG=1 and CONNECTOR.

```
void drm_sysfs_connector_property_event(struct drm_connector *connector, struct  
                                         drm_property *property)
```

generate a DRM uevent for connector property change

Parameters

```
struct drm_connector *connector  
connector on which property changed
```

```
struct drm_property *property  
connector property which has changed.
```

Description

Send a uevent for the specified DRM connector and property. Currently we set HOTPLUG=1 and connector id along with the attached property id related to the change.

```
int drm_class_device_register(struct device *dev)  
register new device with the DRM sysfs class
```

Parameters

```
struct device *dev  
device to register
```

Description

Registers a new struct device within the DRM sysfs class. Essentially only used by ttm to have a place for its global settings. Drivers should never use this.

```
void drm_class_device_unregister(struct device *dev)  
unregister device with the DRM sysfs class
```

Parameters

```
struct device *dev  
device to unregister
```

Description

Unregisters a struct device from the DRM sysfs class. Essentially only used by ttm to have a place for its global settings. Drivers should never use this.

6.10 VBlank event handling

The DRM core exposes two vertical blank related ioctls:

DRM_IOCTL_WAIT_VBLANK

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

DRM_IOCTL_MODESET_CTL

This was only used for user-mode-setting drivers around modesetting changes to allow the kernel to update the vblank interrupt after mode setting, since on many devices the

vertical blank counter is reset to 0 at some point during modeset. Modern drivers should not call this any more since with kernel mode setting it is a no-op.

6.11 Userspace API Structures

DRM exposes many UAPI and structure definition to have a consistent and standardized interface with user. Userspace can refer to these structure definitions and UAPI formats to communicate to driver

6.11.1 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.

DRM_CAP_DUMB_BUFFER

DRM_CAP_DUMB_BUFFER ()

Parameters

Description

If set to 1, the driver supports creating dumb buffers via the DRM_IOCTL_MODE_CREATE_DUMB ioctl.

DRM_CAP_VBLANK_HIGH_CRTC

DRM_CAP_VBLANK_HIGH_CRTC ()

Parameters

Description

If set to 1, the kernel supports specifying a *CRTC index* in the high bits of `drm_wait_vblank_request.type`.

Starting kernel version 2.6.39, this capability is always set to 1.

DRM_CAP_DUMB_PREFERRED_DEPTH

DRM_CAP_DUMB_PREFERRED_DEPTH ()

Parameters

Description

The preferred bit depth for dumb buffers.

The bit depth is the number of bits used to indicate the color of a single pixel excluding any padding. This is different from the number of bits per pixel. For instance, XRGB8888 has a bit depth of 24 but has 32 bits per pixel.

Note that this preference only applies to dumb buffers, it's irrelevant for other types of buffers.

DRM_CAP_DUMB_PREFER_SHADOW

DRM_CAP_DUMB_PREFER_SHADOW ()

Parameters

Description

If set to 1, the driver prefers userspace to render to a shadow buffer instead of directly rendering to a dumb buffer. For best speed, userspace should do streaming ordered memory copies into the dumb buffer and never read from it.

Note that this preference only applies to dumb buffers, it's irrelevant for other types of buffers.

DRM_CAP_PRIME

DRM_CAP_PRIME ()

Parameters

Description

Bitfield of supported PRIME sharing capabilities. See [DRM_PRIME_CAP_IMPORT](#) and [DRM_PRIME_CAP_EXPORT](#).

Starting from kernel version 6.6, both [DRM_PRIME_CAP_IMPORT](#) and [DRM_PRIME_CAP_EXPORT](#) are always advertised.

PRIME buffers are exposed as dma-buf file descriptors. See [PRIME Buffer Sharing](#).

DRM_PRIME_CAP_IMPORT

DRM_PRIME_CAP_IMPORT ()

Parameters

Description

If this bit is set in [DRM_CAP_PRIME](#), the driver supports importing PRIME buffers via the [DRM_IOCTL_PRIME_FD_TO_HANDLE](#) ioctl.

Starting from kernel version 6.6, this bit is always set in [DRM_CAP_PRIME](#).

DRM_PRIME_CAP_EXPORT

DRM_PRIME_CAP_EXPORT ()

Parameters

Description

If this bit is set in [DRM_CAP_PRIME](#), the driver supports exporting PRIME buffers via the [DRM_IOCTL_PRIME_HANDLE_TO_FD](#) ioctl.

Starting from kernel version 6.6, this bit is always set in [DRM_CAP_PRIME](#).

DRM_CAP_TIMESTAMP_MONOTONIC

DRM_CAP_TIMESTAMP_MONOTONIC ()

Parameters

Description

If set to 0, the kernel will report timestamps with `CLOCK_REALTIME` in struct `drm_event_vblank`. If set to 1, the kernel will report timestamps with `CLOCK_MONOTONIC`. See `clock_gettime(2)` for the definition of these clocks.

Starting from kernel version 2.6.39, the default value for this capability is 1. Starting kernel version 4.15, this capability is always set to 1.

DRM_CAP_ASYNC_PAGE_FLIP

`DRM_CAP_ASYNC_PAGE_FLIP` ()

Parameters

Description

If set to 1, the driver supports [`DRM_MODE_PAGE_FLIP_ASYNC`](#).

DRM_CAP_CURSOR_WIDTH

`DRM_CAP_CURSOR_WIDTH` ()

Parameters

Description

The `CURSOR_WIDTH` and `CURSOR_HEIGHT` capabilities return a valid width x height combination for the hardware cursor. The intention is that a hardware agnostic userspace can query a cursor plane size to use.

Note that the cross-driver contract is to merely return a valid size; drivers are free to attach another meaning on top, eg. i915 returns the maximum plane size.

DRM_CAP_CURSOR_HEIGHT

`DRM_CAP_CURSOR_HEIGHT` ()

Parameters

Description

See [`DRM_CAP_CURSOR_WIDTH`](#).

DRM_CAP_ADDFB2_MODIFIERS

`DRM_CAP_ADDFB2_MODIFIERS` ()

Parameters

Description

If set to 1, the driver supports supplying modifiers in the `DRM_IOCTL_MODE_ADDFB2` ioctl.

DRM_CAP_PAGE_FLIP_TARGET

`DRM_CAP_PAGE_FLIP_TARGET` ()

Parameters

Description

If set to 1, the driver supports the `DRM_MODE_PAGE_FLIP_TARGET_ABSOLUTE` and `DRM_MODE_PAGE_FLIP_TARGET_RELATIVE` flags in `drm_mode_crtc_page_flip_target.flags` for the `DRM_IOCTL_MODE_PAGE_FLIP` ioctl.

DRM_CAP_CRTC_IN_VBLANK_EVENT

DRM_CAP_CRTC_IN_VBLANK_EVENT ()

Parameters

Description

If set to 1, the kernel supports reporting the CRTC ID in `drm_event_vblank.crtc_id` for the [DRM_EVENT_VBLANK](#) and [DRM_EVENT_FLIP_COMPLETE](#) events.

Starting kernel version 4.12, this capability is always set to 1.

DRM_CAP_SYNCOBJ

DRM_CAP_SYNCOBJ ()

Parameters

Description

If set to 1, the driver supports sync objects. See [DRM Sync Objects](#).

DRM_CAP_SYNCOBJ_TIMELINE

DRM_CAP_SYNCOBJ_TIMELINE ()

Parameters

Description

If set to 1, the driver supports timeline operations on sync objects. See [DRM Sync Objects](#).

DRM_CLIENT_CAP_STEREO_3D

DRM_CLIENT_CAP_STEREO_3D ()

Parameters

Description

If set to 1, the DRM core will expose the stereo 3D capabilities of the monitor by advertising the supported 3D layouts in the flags of [struct `drm_mode_modeinfo`](#). See `DRM_MODE_FLAG_3D_*`.

This capability is always supported for all drivers starting from kernel version 3.13.

DRM_CLIENT_CAP_UNIVERSAL_PLANES

DRM_CLIENT_CAP_UNIVERSAL_PLANES ()

Parameters

Description

If set to 1, the DRM core will expose all planes (overlay, primary, and cursor) to userspace.

This capability has been introduced in kernel version 3.15. Starting from kernel version 3.17, this capability is always supported for all drivers.

DRM_CLIENT_CAP_ATOMIC

DRM_CLIENT_CAP_ATOMIC ()

Parameters

Description

If set to 1, the DRM core will expose atomic properties to userspace. This implicitly enables [DRM_CLIENT_CAP_UNIVERSAL_PLANES](#) and [DRM_CLIENT_CAP_ASPECT_RATIO](#).

If the driver doesn't support atomic mode-setting, enabling this capability will fail with `-EOPNOTSUPP`.

This capability has been introduced in kernel version 4.0. Starting from kernel version 4.2, this capability is always supported for atomic-capable drivers.

DRM_CLIENT_CAP_ASPECT_RATIO

DRM_CLIENT_CAP_ASPECT_RATIO ()

Parameters

Description

If set to 1, the DRM core will provide aspect ratio information in modes. See [DRM_MODE_FLAG_PIC_AR_*](#).

This capability is always supported for all drivers starting from kernel version 4.18.

DRM_CLIENT_CAP_WRITEBACK_CONNECTORS

DRM_CLIENT_CAP_WRITEBACK_CONNECTORS ()

Parameters

Description

If set to 1, the DRM core will expose special connectors to be used for writing back to memory the scene setup in the commit. The client must enable [DRM_CLIENT_CAP_ATOMIC](#) first.

This capability is always supported for atomic-capable drivers starting from kernel version 4.19.

struct **drm_syncobj_eventfd**

Definition:

```
struct drm_syncobj_eventfd {
    __u32 handle;
    __u32 flags;
    __u64 point;
    __s32 fd;
    __u32 pad;
};
```

Members

handle

syncobj handle.

flags

Zero to wait for the point to be signalled, or `DRM_SYNCOBJ_WAIT_FLAGS_WAIT_AVAILABLE` to wait for a fence to be available for the point.

point

syncobj timeline point (set to zero for binary syncobjs).

fd

Existing eventfd to sent events to.

pad

Must be zero.

Description

Register an eventfd to be signalled by a syncobj. The eventfd counter will be incremented by one.

DRM_IOCTL_GEM_CLOSE

DRM_IOCTL_GEM_CLOSE ()

Close a GEM handle.

Parameters

Description

GEM handles are not reference-counted by the kernel. User-space is responsible for managing their lifetime. For example, if user-space imports the same memory object twice on the same DRM file description, the same GEM handle is returned by both imports, and user-space needs to ensure [DRM_IOCTL_GEM_CLOSE](#) is performed once only. The same situation can happen when a memory object is allocated, then exported and imported again on the same DRM file description. The [DRM_IOCTL_MODE_GETFB2](#) IOCTL is an exception and always returns fresh new GEM handles even if an existing GEM handle already refers to the same memory object before the IOCTL is performed.

DRM_IOCTL_PRIME_HANDLE_TO_FD

DRM_IOCTL_PRIME_HANDLE_TO_FD ()

Convert a GEM handle to a DMA-BUF FD.

Parameters

Description

User-space sets `drm_prime_handle.handle` with the GEM handle to export and `drm_prime_handle.flags`, and gets back a DMA-BUF file descriptor in `drm_prime_handle.fd`.

The export can fail for any driver-specific reason, e.g. because export is not supported for this specific GEM handle (but might be for others).

Support for exporting DMA-BUFs is advertised via [DRM_PRIME_CAP_EXPORT](#).

DRM_IOCTL_PRIME_FD_TO_HANDLE

DRM_IOCTL_PRIME_FD_TO_HANDLE ()

Convert a DMA-BUF FD to a GEM handle.

Parameters

Description

User-space sets `drm_prime_handle.fd` with a DMA-BUF file descriptor to import, and gets back a GEM handle in `drm_prime_handle.handle`. `drm_prime_handle.flags` is unused.

If an existing GEM handle refers to the memory object backing the DMA-BUF, that GEM handle is returned. Therefore user-space which needs to handle arbitrary DMA-BUFs must have a user-space lookup data structure to manually reference-count duplicated GEM handles. For more information see [DRM_IOCTL_GEM_CLOSE](#).

The import can fail for any driver-specific reason, e.g. because import is only supported for DMA-BUFs allocated on this DRM device.

Support for importing DMA-BUFs is advertised via [DRM_PRIME_CAP_IMPORT](#).

DRM_IOCTL_MODE_RMFB

DRM_IOCTL_MODE_RMFB ()

Remove a framebuffer.

Parameters

Description

This removes a framebuffer previously added via ADDFB/ADDFB2. The IOCTL argument is a framebuffer object ID.

Warning: removing a framebuffer currently in-use on an enabled plane will disable that plane. The CRTC the plane is linked to may also be disabled (depending on driver capabilities).

DRM_IOCTL_MODE_GETFB2

DRM_IOCTL_MODE_GETFB2 ()

Get framebuffer metadata.

Parameters

Description

This queries metadata about a framebuffer. User-space fills [drm_mode_fb_cmd2.fb_id](#) as the input, and the kernels fills the rest of the struct as the output.

If the client is DRM master or has CAP_SYS_ADMIN, [drm_mode_fb_cmd2.handles](#) will be filled with GEM buffer handles. Fresh new GEM handles are always returned, even if another GEM handle referring to the same memory object already exists on the DRM file description. The caller is responsible for removing the new handles, e.g. via the [DRM_IOCTL_GEM_CLOSE](#) IOCTL. The same new handle will be returned for multiple planes in case they use the same memory object. Planes are valid until one has a zero handle -- this can be used to compute the number of planes.

Otherwise, [drm_mode_fb_cmd2.handles](#) will be zeroed and planes are valid until one has a zero [drm_mode_fb_cmd2.pitches](#).

If the framebuffer has a format modifier, `DRM_MODE_FB_MODIFIERS` will be set in [drm_mode_fb_cmd2.flags](#) and [drm_mode_fb_cmd2.modifier](#) will contain the modifier. Otherwise, user-space must ignore [drm_mode_fb_cmd2.modifier](#).

To obtain DMA-BUF FDs for each plane without leaking GEM handles, user-space can export each handle via [DRM_IOCTL_PRIME_HANDLE_TO_FD](#), then immediately close each unique handle via [DRM_IOCTL_GEM_CLOSE](#), making sure to not double-close handles which are specified multiple times in the array.

struct **drm_event**

Header for DRM events

Definition:

```
struct drm_event {
    __u32 type;
    __u32 length;
};
```

Members

type

event type.

length

total number of payload bytes (including header).

Description

This struct is a header for events written back to user-space on the DRM FD. A read on the DRM FD will always only return complete events: e.g. if the read buffer is 100 bytes large and there are two 64 byte events pending, only one will be returned.

Event types 0 - 0x7fffffff are generic DRM events, 0x80000000 and up are chipset specific. Generic DRM events include [DRM_EVENT_VBLANK](#), [DRM_EVENT_FLIP_COMPLETE](#) and [DRM_EVENT_CRTC_SEQUENCE](#).

DRM_EVENT_VBLANK

DRM_EVENT_VBLANK ()

vertical blanking event

Parameters

Description

This event is sent in response to `DRM_IOCTL_WAIT_VBLANK` with the `_DRM_VBLANK_EVENT` flag set. The event payload is a struct `drm_event_vblank`.

DRM_EVENT_FLIP_COMPLETE

DRM_EVENT_FLIP_COMPLETE ()

page-flip completion event

Parameters

Description

This event is sent in response to an atomic commit or legacy page-flip with the [DRM_MODE_PAGE_FLIP_EVENT](#) flag set.

The event payload is a struct `drm_event_vblank`.

DRM_EVENT_CRTC_SEQUENCE

DRM_EVENT_CRTC_SEQUENCE ()

CRTC sequence event

Parameters

Description

This event is sent in response to `DRM_IOCTL_CRTC_QUEUE_SEQUENCE`.

The event payload is a struct `drm_event_crtc_sequence`.

struct **drm_mode_modeinfo**

Display mode information.

Definition:

```
struct drm_mode_modeinfo {
    __u32 clock;
    __u16 hdisplay;
    __u16 hsync_start;
    __u16 hsync_end;
    __u16 htotal;
    __u16 hskew;
    __u16 vdisplay;
    __u16 vsync_start;
    __u16 vsync_end;
    __u16 vtotal;
    __u16 vscan;
    __u32 vrefresh;
    __u32 flags;
    __u32 type;
    char name[DRM_DISPLAY_MODE_LEN];
};
```

Members

clock

pixel clock in kHz

hdisplay

horizontal display size

hsync_start

horizontal sync start

hsync_end

horizontal sync end

htotal

horizontal total size

hskew

horizontal skew

vdisplay

vertical display size

vsync_start

vertical sync start

vsync_end

vertical sync end

vtotal

vertical total size

vscan

vertical scan

vrefresh

approximate vertical refresh rate in Hz

flags

bitmask of misc. flags, see `DRM_MODE_FLAG_*` defines

type

bitmask of type flags, see `DRM_MODE_TYPE_*` defines

name

string describing the mode resolution

Description

This is the user-space API display mode information structure. For the kernel version see [*struct drm_display_mode*](#).

struct `drm_mode_get_plane`

Get plane metadata.

Definition:

```
struct drm_mode_get_plane {
    __u32 plane_id;
    __u32 crtc_id;
    __u32 fb_id;
    __u32 possible_crtcs;
    __u32 gamma_size;
    __u32 count_format_types;
    __u64 format_type_ptr;
};
```

Members**plane_id**

Object ID of the plane whose information should be retrieved. Set by caller.

crtc_id

Object ID of the current CRTC.

fb_id

Object ID of the current fb.

possible_crtcs

Bitmask of CRTC's compatible with the plane. CRTC's are created and they receive an index, which corresponds to their position in the bitmask. Bit N corresponds to [*CRTC index*](#) N.

gamma_size

Never used.

count_format_types

Number of formats.

format_type_ptr

Pointer to `__u32` array of formats that are supported by the plane. These formats do not require modifiers.

Description

Userspace can perform a `GETPLANE` ioctl to retrieve information about a plane.

To retrieve the number of formats supported, set **count_format_types** to zero and call the ioctl. **count_format_types** will be updated with the value.

To retrieve these formats, allocate an array with the memory needed to store **count_format_types** formats. Point **format_type_ptr** to this array and call the ioctl again (with **count_format_types** still set to the value returned in the first ioctl call).

struct drm_mode_get_connector

Get connector metadata.

Definition:

```
struct drm_mode_get_connector {
    __u64 encoders_ptr;
    __u64 modes_ptr;
    __u64 props_ptr;
    __u64 prop_values_ptr;
    __u32 count_modes;
    __u32 count_props;
    __u32 count_encoders;
    __u32 encoder_id;
    __u32 connector_id;
    __u32 connector_type;
    __u32 connector_type_id;
    __u32 connection;
    __u32 mm_width;
    __u32 mm_height;
    __u32 subpixel;
    __u32 pad;
};
```

Members**encoders_ptr**

Pointer to `__u32` array of object IDs.

modes_ptr

Pointer to *struct drm_mode_modeinfo* array.

props_ptr

Pointer to `__u32` array of property IDs.

prop_values_ptr

Pointer to `__u64` array of property values.

count_modes

Number of modes.

count_props

Number of properties.

count_encoders

Number of encoders.

encoder_id

Object ID of the current encoder.

connector_id

Object ID of the connector.

connector_type

Type of the connector.

See `DRM_MODE_CONNECTOR_*` defines.

connector_type_id

Type-specific connector number.

This is not an object ID. This is a per-type connector number. Each (type, type_id) combination is unique across all connectors of a DRM device.

The (type, type_id) combination is not a stable identifier: the type_id can change depending on the driver probe order.

connection

Status of the connector.

See [*enum drm_connector_status*](#).

mm_width

Width of the connected sink in millimeters.

mm_height

Height of the connected sink in millimeters.

subpixel

Subpixel order of the connected sink.

See `enum subpixel_order`.

pad

Padding, must be zero.

Description

User-space can perform a `GETCONNECTOR` ioctl to retrieve information about a connector. User-space is expected to retrieve encoders, modes and properties by performing this ioctl at least twice: the first time to retrieve the number of elements, the second time to retrieve the elements themselves.

To retrieve the number of elements, set **count_props** and **count_encoders** to zero, set **count_modes** to 1, and set **modes_ptr** to a temporary *struct drm_mode_modeinfo* element.

To retrieve the elements, allocate arrays for **encoders_ptr**, **modes_ptr**, **props_ptr** and **prop_values_ptr**, then set **count_modes**, **count_props** and **count_encoders** to their capacity.

Performing the ioctl only twice may be racy: the number of elements may have changed with a hotplug event in-between the two ioctls. User-space is expected to retry the last ioctl until the

number of elements stabilizes. The kernel won't fill any array which doesn't have the expected length.

Force-probing a connector

If the **count_modes** field is set to zero and the DRM client is the current DRM master, the kernel will perform a forced probe on the connector to refresh the connector status, modes and EDID. A forced-probe can be slow, might cause flickering and the ioctl will block.

User-space needs to force-probe connectors to ensure their metadata is up-to-date at startup and after receiving a hot-plug event. User-space may perform a forced-probe when the user explicitly requests it. User-space shouldn't perform a forced-probe in other situations.

struct **drm_mode_property_enum**

Description for an enum/bitfield entry.

Definition:

```
struct drm_mode_property_enum {
    __u64 value;
    char name[DRM_PROP_NAME_LEN];
};
```

Members

value

numeric value for this enum entry.

name

symbolic name for this enum entry.

Description

See *struct drm_property_enum* for details.

struct **drm_mode_get_property**

Get property metadata.

Definition:

```
struct drm_mode_get_property {
    __u64 values_ptr;
    __u64 enum_blob_ptr;
    __u32 prop_id;
    __u32 flags;
    char name[DRM_PROP_NAME_LEN];
    __u32 count_values;
    __u32 count_enum_blobs;
};
```

Members

values_ptr

Pointer to a __u64 array.

enum_blob_ptr

Pointer to a *struct drm_mode_property_enum* array.

prop_id

Object ID of the property which should be retrieved. Set by the caller.

flags

DRM_MODE_PROP_* bitfield. See [drm_property.flags](#) for a definition of the flags.

name

Symbolic property name. User-space should use this field to recognize properties.

count_values

Number of elements in **values_ptr**.

count_enum_blobs

Number of elements in **enum_blob_ptr**.

Description

User-space can perform a GETPROPERTY ioctl to retrieve information about a property. The same property may be attached to multiple objects, see “Modeset Base Object Abstraction”.

The meaning of the **values_ptr** field changes depending on the property type. See [drm_property.flags](#) for more details.

The **enum_blob_ptr** and **count_enum_blobs** fields are only meaningful when the property has the type DRM_MODE_PROP_ENUM or DRM_MODE_PROP_BITMASK. For backwards compatibility, the kernel will always set **count_enum_blobs** to zero when the property has the type DRM_MODE_PROP_BLOB. User-space must ignore these two fields if the property has a different type.

User-space is expected to retrieve values and enums by performing this ioctl at least twice: the first time to retrieve the number of elements, the second time to retrieve the elements themselves.

To retrieve the number of elements, set **count_values** and **count_enum_blobs** to zero, then call the ioctl. **count_values** will be updated with the number of elements. If the property has the type DRM_MODE_PROP_ENUM or DRM_MODE_PROP_BITMASK, **count_enum_blobs** will be updated as well.

To retrieve the elements themselves, allocate an array for **values_ptr** and set **count_values** to its capacity. If the property has the type DRM_MODE_PROP_ENUM or DRM_MODE_PROP_BITMASK, allocate an array for **enum_blob_ptr** and set **count_enum_blobs** to its capacity. Calling the ioctl again will fill the arrays.

struct drm_mode_fb_cmd2

Frame-buffer metadata.

Definition:

```
struct drm_mode_fb_cmd2 {
    __u32 fb_id;
    __u32 width;
    __u32 height;
    __u32 pixel_format;
    __u32 flags;
    __u32 handles[4];
    __u32 pitches[4];
    __u32 offsets[4];
}
```

```
__u64 modifier[4];
};
```

Members

fb_id

Object ID of the frame-buffer.

width

Width of the frame-buffer.

height

Height of the frame-buffer.

pixel_format

FourCC format code, see `DRM_FORMAT_*` constants in `drm_fourcc.h`.

flags

Frame-buffer flags (see `DRM_MODE_FB_INTERLACED` and `DRM_MODE_FB_MODIFIERS`).

handles

GEM buffer handle, one per plane. Set to 0 if the plane is unused. The same handle can be used for multiple planes.

pitches

Pitch (aka. stride) in bytes, one per plane.

offsets

Offset into the buffer in bytes, one per plane.

modifier

Format modifier, one per plane. See `DRM_FORMAT_MOD_*` constants in `drm_fourcc.h`. All planes must use the same modifier. Ignored unless `DRM_MODE_FB_MODIFIERS` is set in **flags**.

Description

This struct holds frame-buffer metadata. There are two ways to use it:

- User-space can fill this struct and perform a `DRM_IOCTL_MODE_ADDFB2` ioctl to register a new frame-buffer. The new frame-buffer object ID will be set by the kernel in **fb_id**.
- User-space can set **fb_id** and perform a `DRM_IOCTL_MODE_GETFB2` ioctl to fetch metadata about an existing frame-buffer.

In case of planar formats, this struct allows up to 4 buffer objects with offsets and pitches per plane. The pitch and offset order are dictated by the format FourCC as defined by `drm_fourcc.h`, e.g. NV12 is described as:

YUV 4:2:0 image with a plane of 8-bit Y samples followed by an interleaved U/V plane containing 8-bit 2x2 subsampled colour difference samples.

So it would consist of a Y plane at `offsets[0]` and a UV plane at `offsets[1]`.

To accommodate tiled, compressed, etc formats, a modifier can be specified. For more information see the “Format Modifiers” section. Note that even though it looks like we have a modifier per-plane, we in fact do not. The modifier for each plane must be identical. Thus all combinations of different data layouts for multi-plane formats must be enumerated as separate modifiers.

All of the entries in **handles**, **pitches**, **offsets** and **modifier** must be zero when unused. Warning, for **offsets** and **modifier** zero can't be used to figure out whether the entry is used or not since it's a valid value (a zero offset is common, and a zero modifier is `DRM_FORMAT_MOD_LINEAR`).

struct **hdr_metadata_infotrame**

HDR Metadata Infotrame Data.

Definition:

```
struct hdr_metadata_infotrame {
    __u8 eotf;
    __u8 metadata_type;
    struct {
        __u16 x, y;
    } display primaries[3];
    struct {
        __u16 x, y;
    } white_point;
    __u16 max_display_mastering_luminance;
    __u16 min_display_mastering_luminance;
    __u16 max_cll;
    __u16 max_fall;
};
```

Members

eotf

Electro-Optical Transfer Function (EOTF) used in the stream.

metadata_type

Static_Metadata_Descriptor_ID.

display_primaries

Color Primaries of the Data. These are coded as unsigned 16-bit values in units of 0.00002, where 0x0000 represents zero and 0xC350 represents 1.0000. **display_primaries.x**: X coordinate of color primary. **display_primaries.y**: Y coordinate of color primary.

white_point

White Point of Colorspace Data. These are coded as unsigned 16-bit values in units of 0.00002, where 0x0000 represents zero and 0xC350 represents 1.0000. **white_point.x**: X coordinate of whitepoint of color primary. **white_point.y**: Y coordinate of whitepoint of color primary.

max_display_mastering_luminance

Max Mastering Display Luminance. This value is coded as an unsigned 16-bit value in units of 1 cd/m², where 0x0001 represents 1 cd/m² and 0xFFFF represents 65535 cd/m².

min_display_mastering_luminance

Min Mastering Display Luminance. This value is coded as an unsigned 16-bit value in units of 0.0001 cd/m², where 0x0001 represents 0.0001 cd/m² and 0xFFFF represents 6.5535 cd/m².

max_cll

Max Content Light Level. This value is coded as an unsigned 16-bit value in units of 1 cd/m², where 0x0001 represents 1 cd/m² and 0xFFFF represents 65535 cd/m².

max_fall

Max Frame Average Light Level. This value is coded as an unsigned 16-bit value in units of 1 cd/m2, where 0x0001 represents 1 cd/m2 and 0xFFFF represents 65535 cd/m2.

Description

HDR Metadata Infoframe as per CTA 861.G spec. This is expected to match exactly with the spec.

Userspace is expected to pass the metadata information as per the format described in this structure.

struct hdr_output_metadata

HDR output metadata

Definition:

```
struct hdr_output_metadata {
    __u32 metadata_type;
    union {
        struct hdr_metadata_infoframe hdmi_metadata_type1;
    };
};
```

Members**metadata_type**

Static_Metadata_Descriptor_ID.

{unnamed_union}

anonymous

hdmi_metadata_type1

HDR Metadata Infoframe.

Description

Metadata Information to be passed from userspace

DRM_MODE_PAGE_FLIP_EVENT

DRM_MODE_PAGE_FLIP_EVENT ()

Parameters**Description**

Request that the kernel sends back a vblank event (see struct `drm_event_vblank`) with the [DRM_EVENT_FLIP_COMPLETE](#) type when the page-flip is done.

DRM_MODE_PAGE_FLIP_ASYNC

DRM_MODE_PAGE_FLIP_ASYNC ()

Parameters**Description**

Request that the page-flip is performed as soon as possible, ie. with no delay due to waiting for vblank. This may cause tearing to be visible on the screen.

DRM_MODE_PAGE_FLIP_FLAGS

DRM_MODE_PAGE_FLIP_FLAGS ()

Parameters

Description

Bitmask of flags suitable for `drm_mode_crtc_page_flip_target.flags`.

DRM_MODE_ATOMIC_TEST_ONLY

DRM_MODE_ATOMIC_TEST_ONLY ()

Parameters

Description

Do not apply the atomic commit, instead check whether the hardware supports this configuration.

See [*drm_mode_config_funcs.atomic_check*](#) for more details on test-only commits.

DRM_MODE_ATOMIC_NONBLOCK

DRM_MODE_ATOMIC_NONBLOCK ()

Parameters

Description

Do not block while applying the atomic commit. The `DRM_IOCTL_MODE_ATOMIC` IOCTL returns immediately instead of waiting for the changes to be applied in hardware. Note, the driver will still check that the update can be applied before retuning.

DRM_MODE_ATOMIC_ALLOW_MODESET

DRM_MODE_ATOMIC_ALLOW_MODESET ()

Parameters

Description

Allow the update to result in temporary or transient visible artifacts while the update is being applied. Applying the update may also take significantly more time than a page flip. All visual artifacts will disappear by the time the update is completed, as signalled through the vblank event's timestamp (see `struct drm_event_vblank`).

This flag must be set when the KMS update might cause visible artifacts. Without this flag such KMS update will return a `EINVAL` error. What kind of update may cause visible artifacts depends on the driver and the hardware. User-space that needs to know beforehand if an update might cause visible artifacts can use [*DRM_MODE_ATOMIC_TEST_ONLY*](#) without [*DRM_MODE_ATOMIC_ALLOW_MODESET*](#) to see if it fails.

To the best of the driver's knowledge, visual artifacts are guaranteed to not appear when this flag is not set. Some sinks might display visual artifacts outside of the driver's control.

DRM_MODE_ATOMIC_FLAGS

DRM_MODE_ATOMIC_FLAGS ()

Parameters

Description

Bitfield of flags accepted by the `DRM_IOCTL_MODE_ATOMIC` IOCTL in `drm_mode_atomic.flags`.

struct **drm_mode_create_blob**

Create New blob property

Definition:

```
struct drm_mode_create_blob {
    __u64 data;
    __u32 length;
    __u32 blob_id;
};
```

Members

data

Pointer to data to copy.

length

Length of data to copy.

blob_id

Return: new property ID.

Description

Create a new 'blob' data property, copying length bytes from data pointer, and returning new blob ID.

struct **drm_mode_destroy_blob**

Destroy user blob

Definition:

```
struct drm_mode_destroy_blob {
    __u32 blob_id;
};
```

Members

blob_id

blob_id to destroy

Description

Destroy a user-created blob property.

User-space can release blobs as soon as they do not need to refer to them by their blob object ID. For instance, if you are using a `MODE_ID` blob in an atomic commit and you will not make another commit re-using the same ID, you can destroy the blob as soon as the commit has been issued, without waiting for it to complete.

struct **drm_mode_create_lease**

Create lease

Definition:

```
struct drm_mode_create_lease {
    __u64 object_ids;
    __u32 object_count;
    __u32 flags;
    __u32 lessee_id;
    __u32 fd;
};
```

Members

object_ids

Pointer to array of object ids (__u32)

object_count

Number of object ids

flags

flags for new FD (O_CLOEXEC, etc)

lessee_id

Return: unique identifier for lessee.

fd

Return: file descriptor to new drm_master file

Description

Lease mode resources, creating another drm_master.

The **object_ids** array must reference at least one CRTC, one connector and one plane if [*DRM_CLIENT_CAP_UNIVERSAL_PLANES*](#) is enabled. Alternatively, the lease can be completely empty.

struct **drm_mode_list_lessees**

List lessees

Definition:

```
struct drm_mode_list_lessees {
    __u32 count_lessees;
    __u32 pad;
    __u64 lessees_ptr;
};
```

Members

count_lessees

Number of lessees.

On input, provides length of the array. On output, provides total number. No more than the input number will be written back, so two calls can be used to get the size and then the data.

pad

Padding.

lessees_ptr

Pointer to lessees.

Pointer to __u64 array of lessee ids

Description

List lessees from a drm_master.

struct **drm_mode_get_lease**

Get Lease

Definition:

```
struct drm_mode_get_lease {
    __u32 count_objects;
    __u32 pad;
    __u64 objects_ptr;
};
```

Members

count_objects

Number of leased objects.

On input, provides length of the array. On output, provides total number. No more than the input number will be written back, so two calls can be used to get the size and then the data.

pad

Padding.

objects_ptr

Pointer to objects.

Pointer to __u32 array of object ids.

Description

Get leased objects.

struct **drm_mode_revoke_lease**

Revoke lease

Definition:

```
struct drm_mode_revoke_lease {
    __u32 lessee_id;
};
```

Members

lessee_id

Unique ID of lessee

struct **drm_mode_rect**

Two dimensional rectangle.

Definition:

```
struct drm_mode_rect {
    __s32 x1;
```

```
__s32 y1;  
__s32 x2;  
__s32 y2;  
};
```

Members

x1

Horizontal starting coordinate (inclusive).

y1

Vertical starting coordinate (inclusive).

x2

Horizontal ending coordinate (exclusive).

y2

Vertical ending coordinate (exclusive).

Description

With drm subsystem using *struct drm_rect* to manage rectangular area this export it to user-space.

Currently used by `drm_mode_atomic` blob property `FB_DAMAGE_CLIPS`.

DRM CLIENT USAGE STATS

DRM drivers can choose to export partly standardised text output via the *fops->show_fdinfo()* as part of the driver specific file operations registered in the `struct drm_driver` object registered with the DRM core.

One purpose of this output is to enable writing as generic as practically feasible *top(1)* like userspace monitoring tools.

Given the differences between various DRM drivers the specification of the output is split between common and driver specific parts. Having said that, wherever possible effort should still be made to standardise as much as possible.

7.1 File format specification

- File shall contain one key value pair per one line of text.
- Colon character (:) must be used to delimit keys and values.
- All keys shall be prefixed with *drm-*.
- Whitespace between the delimiter and first non-whitespace character shall be ignored when parsing.
- Keys are not allowed to contain whitespace characters.
- Numerical key value pairs can end with optional unit string.
- Data type of the value is fixed as defined in the specification.

7.1.1 Key types

1. Mandatory, fully standardised.
2. Optional, fully standardised.
3. Driver specific.

7.1.2 Data types

- `<uint>` - Unsigned integer without defining the maximum value.
- `<keystr>` - String excluding any above defined reserved characters or whitespace.
- `<valstr>` - String.

7.1.3 Mandatory fully standardised keys

- `drm-driver: <valstr>`

String shall contain the name this driver registered as via the respective `struct drm_driver` data structure.

7.1.4 Optional fully standardised keys

Identification

- `drm-pdev: <aaaa:bb.cc.d>`

For PCI devices this should contain the PCI slot address of the device in question.

- `drm-client-id: <uint>`

Unique value relating to the open DRM file descriptor used to distinguish duplicated and shared file descriptors. Conceptually the value should map 1:1 to the in kernel representation of `struct drm_file` instances.

Uniqueness of the value shall be either globally unique, or unique within the scope of each device, in which case `drm-pdev` shall be present as well.

Userspace should make sure to not double account any usage statistics by using the above described criteria in order to associate data to individual clients.

Utilization

- `drm-engine-<keystr>: <uint> ns`

GPUs usually contain multiple execution engines. Each shall be given a stable and unique name (`keystr`), with possible values documented in the driver specific documentation.

Value shall be in specified time units which the respective GPU engine spent busy executing workloads belonging to this client.

Values are not required to be constantly monotonic if it makes the driver implementation easier, but are required to catch up with the previously reported larger value within a reasonable period. Upon observing a value lower than what was previously read, userspace is expected to stay with that larger previous value until a monotonic update is seen.

- `drm-engine-capacity-<keystr>: <uint>`

Engine identifier string must be the same as the one specified in the `drm-engine-<keystr>` tag and shall contain a greater than zero number in case the exported engine corresponds to a group of identical hardware engines.

In the absence of this tag parser shall assume capacity of one. Zero capacity is not allowed.

- `drm-cycles-<keyst>: <uint>`

Engine identifier string must be the same as the one specified in the `drm-engine-<keyst>` tag and shall contain the number of busy cycles for the given engine.

Values are not required to be constantly monotonic if it makes the driver implementation easier, but are required to catch up with the previously reported larger value within a reasonable period. Upon observing a value lower than what was previously read, userspace is expected to stay with that larger previous value until a monotonic update is seen.

- `drm-maxfreq-<keyst>: <uint> [Hz|MHz|KHz]`

Engine identifier string must be the same as the one specified in the `drm-engine-<keyst>` tag and shall contain the maximum frequency for the given engine. Taken together with `drm-cycles-<keyst>`, this can be used to calculate percentage utilization of the engine, whereas `drm-engine-<keyst>` only reflects time active without considering what frequency the engine is operating as a percentage of its maximum frequency.

Memory

- `drm-memory-<region>: <uint> [KiB|MiB]`

Each possible memory type which can be used to store buffer objects by the GPU in question shall be given a stable and unique name to be returned as the string here. The name “memory” is reserved to refer to normal system memory.

Value shall reflect the amount of storage currently consumed by the buffer objects belong to this client, in the respective memory region.

Default unit shall be bytes with optional unit specifiers of ‘KiB’ or ‘MiB’ indicating kibi- or mebi-bytes.

- `drm-shared-<region>: <uint> [KiB|MiB]`

The total size of buffers that are shared with another file (ie. have more than a single handle).

- `drm-total-<region>: <uint> [KiB|MiB]`

The total size of buffers that including shared and private memory.

- `drm-resident-<region>: <uint> [KiB|MiB]`

The total size of buffers that are resident in the specified region.

- `drm-purgeable-<region>: <uint> [KiB|MiB]`

The total size of buffers that are purgeable.

- `drm-active-<region>: <uint> [KiB|MiB]`

The total size of buffers that are active on one or more engines.

7.2 Implementation Details

Drivers should use `drm_show_fdinfo()` in their *struct file_operations*, and implement `&drm_driver.show_fdinfo` if they wish to provide any stats which are not provided by `drm_show_fdinfo()`. But even driver specific stats should be documented above and where possible, aligned with other drivers.

7.2.1 Driver specific implementations

i915 DRM client usage stats implementation

DRM DRIVER UAPI

8.1 drm/i915 uAPI

uevents generated by i915 on it's device node

I915_L3_PARITY_UEVENT - Generated when the driver receives a parity mismatch event from the gpu l3 cache. Additional information supplied is ROW, BANK, SUBBANK, SLICE of the affected cacheline. Userspace should keep track of these events and if a specific cache-line seems to have a persistent error remap it with the l3 remapping tool supplied in intel-gpu-tools. The value supplied with the event is always 1.

I915_ERROR_UEVENT - Generated upon error detection, currently only via hangcheck. The error detection event is a good indicator of when things began to go badly. The value supplied with the event is a 1 upon error detection, and a 0 upon reset completion, signifying no more error exists. NOTE: Disabling hangcheck or reset via module parameter will cause the related events to not be seen.

I915_RESET_UEVENT - Event is generated just before an attempt to reset the GPU. The value supplied with the event is always 1. NOTE: Disable reset via module parameter will cause this event to not be seen.

struct i915_user_extension

Base class for defining a chain of extensions

Definition:

```
struct i915_user_extension {
    __u64 next_extension;
    __u32 name;
    __u32 flags;
    __u32 rsvd[4];
};
```

Members

next_extension

Pointer to the next *struct i915_user_extension*, or zero if the end.

name

Name of the extension.

Note that the name here is just some integer.

Also note that the name space for this is not global for the whole driver, but rather its scope/meaning is limited to the specific piece of uAPI which has embedded the *struct i915_user_extension*.

flags

MBZ

All undefined bits must be zero.

rsvd

MBZ

Reserved for future use; must be zero.

Description

Many interfaces need to grow over time. In most cases we can simply extend the struct and have userspace pass in more data. Another option, as demonstrated by Vulkan's approach to providing extensions for forward and backward compatibility, is to use a list of optional structs to provide those extra details.

The key advantage to using an extension chain is that it allows us to redefine the interface more easily than an ever growing struct of increasing complexity, and for large parts of that interface to be entirely optional. The downside is more pointer chasing; chasing across the __user boundary with pointers encapsulated inside u64.

Example chaining:

```
struct i915_user_extension ext3 {
    .next_extension = 0, // end
    .name = ...,
};
struct i915_user_extension ext2 {
    .next_extension = (uintptr_t)&ext3,
    .name = ...,
};
struct i915_user_extension ext1 {
    .next_extension = (uintptr_t)&ext2,
    .name = ...,
};
```

Typically the *struct i915_user_extension* would be embedded in some uAPI struct, and in this case we would feed it the head of the chain(i.e ext1), which would then apply all of the above extensions.

enum **drm_i915_gem_engine_class**

uapi engine type enumeration

Constants

I915_ENGINE_CLASS_RENDER

Render engines support instructions used for 3D, Compute (GPGPU), and programmable media workloads. These instructions fetch data and dispatch individual work items to threads that operate in parallel. The threads run small programs (called "kernels" or "shaders") on the GPU's execution units (EUs).

I915_ENGINE_CLASS_COPY

Copy engines (also referred to as “blitters”) support instructions that move blocks of data from one location in memory to another, or that fill a specified location of memory with fixed data. Copy engines can perform pre-defined logical or bitwise operations on the source, destination, or pattern data.

I915_ENGINE_CLASS_VIDEO

Video engines (also referred to as “bit stream decode” (BSD) or “vdbox”) support instructions that perform fixed-function media decode and encode.

I915_ENGINE_CLASS_VIDEO_ENHANCE

Video enhancement engines (also referred to as “vebox”) support instructions related to image enhancement.

I915_ENGINE_CLASS_COMPUTE

Compute engines support a subset of the instructions available on render engines: compute engines support Compute (GPGPU) and programmable media workloads, but do not support the 3D pipeline.

I915_ENGINE_CLASS_INVALID

Placeholder value to represent an invalid engine class assignment.

Description

Different engines serve different roles, and there may be more than one engine serving each role. This enum provides a classification of the role of the engine, which may be used when requesting operations to be performed on a certain subset of engines, or for providing information about that group.

struct **i915_engine_class_instance**

Engine class/instance identifier

Definition:

```
struct i915_engine_class_instance {
    __u16 engine_class;
#define I915_ENGINE_CLASS_INVALID_NONE -1;
#define I915_ENGINE_CLASS_INVALID_VIRTUAL -2;
    __u16 engine_instance;
};
```

Members**engine_class**

Engine class from [enum `drm_i915_gem_engine_class`](#)

engine_instance

Engine instance.

Description

There may be more than one engine fulfilling any role within the system. Each engine of a class is given a unique instance number and therefore any engine can be specified by its class:instance tuple. APIs that allow access to any engine in the system will use [struct `i915_engine_class_instance`](#) for this identification.

perf_events exposed by i915 through `/sys/bus/event_sources/drivers/i915`

struct **drm_i915_getparam**

Driver parameter query structure.

Definition:

```
struct drm_i915_getparam {
    __s32 param;
    int __user *value;
};
```

Members

param

Driver parameter to query.

value

Address of memory where queried value should be put.

WARNING: Using pointers instead of fixed-size u64 means we need to write compat32 code. Don't repeat this mistake.

type **drm_i915_getparam_t**

Driver parameter query structure. See *struct drm_i915_getparam*.

struct **drm_i915_gem_mmap_offset**

Retrieve an offset so we can mmap this buffer object.

Definition:

```
struct drm_i915_gem_mmap_offset {
    __u32 handle;
    __u32 pad;
    __u64 offset;
    __u64 flags;
#define I915_MMAP_OFFSET_GTT    0;
#define I915_MMAP_OFFSET_WC    1;
#define I915_MMAP_OFFSET_WB    2;
#define I915_MMAP_OFFSET_UC    3;
#define I915_MMAP_OFFSET_FIXED 4;
    __u64 extensions;
};
```

Members

handle

Handle for the object being mapped.

pad

Must be zero

offset

The fake offset to use for subsequent mmap call

This is a fixed-size type for 32/64 compatibility.

flags

Flags for extended behaviour.

It is mandatory that one of the *MMAP_OFFSET* types should be included:

- *I915_MMAP_OFFSET_GTT*: Use mmap with the object bound to GTT. (Write-Combined)
- *I915_MMAP_OFFSET_WC*: Use Write-Combined caching.
- *I915_MMAP_OFFSET_WB*: Use Write-Back caching.
- *I915_MMAP_OFFSET_FIXED*: Use object placement to determine caching.

On devices with local memory *I915_MMAP_OFFSET_FIXED* is the only valid type. On devices without local memory, this caching mode is invalid.

As caching mode when specifying *I915_MMAP_OFFSET_FIXED*, WC or WB will be used, depending on the object placement on creation. WB will be used when the object can only exist in system memory, WC otherwise.

extensions

Zero-terminated chain of extensions.

No current extensions defined; mbz.

Description

This struct is passed as argument to the *DRM_IOCTL_I915_GEM_MMAP_OFFSET* ioctl, and is used to retrieve the fake offset to mmap an object specified by handle.

The legacy way of using *DRM_IOCTL_I915_GEM_MMAP* is removed on gen12+. *DRM_IOCTL_I915_GEM_MMAP_GTT* is an older supported alias to this struct, but will behave as setting the extensions to 0, and flags to *I915_MMAP_OFFSET_GTT*.

struct **drm_i915_gem_set_domain**

Adjust the objects write or read domain, in preparation for accessing the pages via some CPU domain.

Definition:

```
struct drm_i915_gem_set_domain {
    __u32 handle;
    __u32 read_domains;
    __u32 write_domain;
};
```

Members

handle

Handle for the object.

read_domains

New read domains.

write_domain

New write domain.

Note that having something in the write domain implies it's in the read domain, and only that read domain.

Description

Specifying a new write or read domain will flush the object out of the previous domain(if required), before then updating the objects domain tracking with the new domain.

Note this might involve waiting for the object first if it is still active on the GPU.

Supported values for **read_domains** and **write_domain**:

- I915_GEM_DOMAIN_WC: Uncached write-combined domain
- I915_GEM_DOMAIN_CPU: CPU cache domain
- I915_GEM_DOMAIN_GTT: Mappable aperture domain

All other domains are rejected.

Note that for discrete, starting from DG1, this is no longer supported, and is instead rejected. On such platforms the CPU domain is effectively static, where we also only support a single [drm_i915_gem_mmap_offset](#) cache mode, which can't be set explicitly and instead depends on the object placements, as per the below.

Implicit caching rules, starting from DG1:

- If any of the object placements (see [drm_i915_gem_create_ext_memory_regions](#)) contain I915_MEMORY_CLASS_DEVICE then the object will be allocated and mapped as write-combined only.
- Everything else is always allocated and mapped as write-back, with the guarantee that everything is also coherent with the GPU.

Note that this is likely to change in the future again, where we might need more flexibility on future devices, so making this all explicit as part of a new [drm_i915_gem_create_ext](#) extension is probable.

struct **drm_i915_gem_exec_fence**

An input or output fence for the execbuf ioctl.

Definition:

```
struct drm_i915_gem_exec_fence {
    __u32 handle;
    __u32 flags;
#define I915_EXEC_FENCE_WAIT          (1<<0);
#define I915_EXEC_FENCE_SIGNAL       (1<<1);
#define __I915_EXEC_FENCE_UNKNOWN_FLAGS  (- (I915_EXEC_FENCE_SIGNAL << 1));
};
```

Members

handle

User's handle for a `drm_syncobj` to wait on or signal.

flags

Supported flags are:

I915_EXEC_FENCE_WAIT: Wait for the input fence before request submission.

I915_EXEC_FENCE_SIGNAL: Return request completion fence as output

Description

The request will wait for input fence to signal before submission.

The returned output fence will be signaled after the completion of the request.

struct **drm_i915_gem_execbuffer_ext_timeline_fences**

Timeline fences for execbuf ioctl.

Definition:

```
struct drm_i915_gem_execbuffer_ext_timeline_fences {
#define DRM_I915_GEM_EXECBUFFER_EXT_TIMELINE_FENCES 0;
    struct i915_user_extension base;
    __u64 fence_count;
    __u64 handles_ptr;
    __u64 values_ptr;
};
```

Members

base

Extension link. See *struct i915_user_extension*.

fence_count

Number of elements in the **handles_ptr** & **value_ptr** arrays.

handles_ptr

Pointer to an array of *struct drm_i915_gem_exec_fence* of length **fence_count**.

values_ptr

Pointer to an array of u64 values of length **fence_count**. Values must be 0 for a binary drm_syncobj. A Value of 0 for a timeline drm_syncobj is invalid as it turns a drm_syncobj into a binary one.

Description

This structure describes an array of drm_syncobj and associated points for timeline variants of drm_syncobj. It is invalid to append this structure to the execbuf if I915_EXEC_FENCE_ARRAY is set.

struct **drm_i915_gem_execbuffer2**

Structure for DRM_I915_GEM_EXECBUFFER2 ioctl.

Definition:

```
struct drm_i915_gem_execbuffer2 {
    __u64 buffers_ptr;
    __u32 buffer_count;
    __u32 batch_start_offset;
    __u32 batch_len;
    __u32 DR1;
    __u32 DR4;
    __u32 num_cliprects;
    __u64 cliprects_ptr;
    __u64 flags;
#define I915_EXEC_RING_MASK                (0x3f);
#define I915_EXEC_DEFAULT                  (0<<0);
#define I915_EXEC_RENDER                   (1<<0);
#define I915_EXEC_BSD                      (2<<0);
```

```
#define I915_EXEC_BLT (3<<0);
#define I915_EXEC_VEB0X (4<<0);
#define I915_EXEC_CONSTANTS_MASK (3<<6);
#define I915_EXEC_CONSTANTS_REL_GENERAL (0<<6);
#define I915_EXEC_CONSTANTS_ABSOLUTE (1<<6);
#define I915_EXEC_CONSTANTS_REL_SURFACE (2<<6);
#define I915_EXEC_GEN7_SOL_RESET (1<<8);
#define I915_EXEC_SECURE (1<<9);
#define I915_EXEC_IS_PINNED (1<<10);
#define I915_EXEC_NO_RELOC (1<<11);
#define I915_EXEC_HANDLE_LUT (1<<12);
#define I915_EXEC_BSD_SHIFT (13);
#define I915_EXEC_BSD_MASK (3 << I915_EXEC_BSD_SHIFT);
#define I915_EXEC_BSD_DEFAULT (0 << I915_EXEC_BSD_SHIFT);
#define I915_EXEC_BSD_RING1 (1 << I915_EXEC_BSD_SHIFT);
#define I915_EXEC_BSD_RING2 (2 << I915_EXEC_BSD_SHIFT);
#define I915_EXEC_RESOURCE_STREAMER (1<<15);
#define I915_EXEC_FENCE_IN (1<<16);
#define I915_EXEC_FENCE_OUT (1<<17);
#define I915_EXEC_BATCH_FIRST (1<<18);
#define I915_EXEC_FENCE_ARRAY (1<<19);
#define I915_EXEC_FENCE_SUBMIT (1 << 20);
#define I915_EXEC_USE_EXTENSIONS (1 << 21);
#define __I915_EXEC_UNKNOWN_FLAGS (- (I915_EXEC_USE_EXTENSIONS << 1));
    __u64 rsvd1;
    __u64 rsvd2;
};
```

Members

buffers_ptr

Pointer to a list of `gem_exec_object2` structs

buffer_count

Number of elements in **buffers_ptr** array

batch_start_offset

Offset in the batchbuffer to start execution from.

batch_len

Length in bytes of the batch buffer, starting from the **batch_start_offset**. If 0, length is assumed to be the batch buffer object size.

DR1

deprecated

DR4

deprecated

num_cliprects

See **cliprects_ptr**

cliprects_ptr

Kernel clipping was a DRI1 misfeature.

It is invalid to use this field if `I915_EXEC_FENCE_ARRAY` or `I915_EXEC_USE_EXTENSIONS` flags are not set.

If `I915_EXEC_FENCE_ARRAY` is set, then this is a pointer to an array of [`drm_i915_gem_exec_fence`](#) and **num_cliprects** is the length of the array.

If `I915_EXEC_USE_EXTENSIONS` is set, then this is a pointer to a single [`i915_user_extension`](#) and `num_cliprects` is 0.

flags

Execbuf flags

rsvd1

Context id

rsvd2

in and out `sync_file` file descriptors.

When `I915_EXEC_FENCE_IN` or `I915_EXEC_FENCE_SUBMIT` flag is set, the lower 32 bits of this field will have the in `sync_file` fd (input).

When `I915_EXEC_FENCE_OUT` flag is set, the upper 32 bits of this field will have the out `sync_file` fd (output).

struct `drm_i915_gem_caching`

Set or get the caching for given object handle.

Definition:

```
struct drm_i915_gem_caching {
    __u32 handle;
#define I915_CACHING_NONE          0;
#define I915_CACHING_CACHED       1;
#define I915_CACHING_DISPLAY     2;
    __u32 caching;
};
```

Members

handle

Handle of the buffer to set/get the caching level.

caching

The GTT caching level to apply or possible return value.

The supported **caching** values:

`I915_CACHING_NONE`:

GPU access is not coherent with CPU caches. Default for machines without an LLC. This means manual flushing might be needed, if we want GPU access to be coherent.

`I915_CACHING_CACHED`:

GPU access is coherent with CPU caches and furthermore the data is cached in last-level caches shared between CPU cores and the GPU GT.

`I915_CACHING_DISPLAY`:

Special GPU caching mode which is coherent with the scanout engines. Transparently falls back to `I915_CACHING_NONE` on platforms where no special cache mode (like write-through or gfdt flushing) is available. The kernel automatically sets this mode when using a buffer as a scanout target. Userspace can manually set this mode to avoid a costly stall and `clflush` in the hotpath of drawing the first frame.

Description

Allow userspace to control the GTT caching bits for a given object when the object is later mapped through the `ppGTT`(or `GGTT` on older platforms lacking `ppGTT` support, or if the object is used for scanout). Note that this might require unbinding the object from the GTT first, if its current caching value doesn't match.

Note that this all changes on discrete platforms, starting from DG1, the set/get caching is no longer supported, and is now rejected. Instead the CPU caching attributes(`WB` vs `WC`) will become an immutable creation time property for the object, along with the GTT caching level. For now we don't expose any new uAPI for this, instead on DG1 this is all implicit, although this largely shouldn't matter since DG1 is coherent by default(without any way of controlling it).

Implicit caching rules, starting from DG1:

- If any of the object placements (see [`drm_i915_gem_create_ext_memory_regions`](#)) contain `I915_MEMORY_CLASS_DEVICE` then the object will be allocated and mapped as write-combined only.
- Everything else is always allocated and mapped as write-back, with the guarantee that everything is also coherent with the GPU.

Note that this is likely to change in the future again, where we might need more flexibility on future devices, so making this all explicit as part of a new [`drm_i915_gem_create_ext`](#) extension is probable.

Side note: Part of the reason for this is that changing the at-allocation-time CPU caching attributes for the pages might be required(and is expensive) if we need to then CPU map the pages later with different caching attributes. This inconsistent caching behaviour, while supported on x86, is not universally supported on other architectures. So for simplicity we opt for setting everything at creation time, whilst also making it immutable, on discrete platforms.

struct **drm_i915_gem_context_create_ext**

Structure for creating contexts.

Definition:

```
struct drm_i915_gem_context_create_ext {
    __u32 ctx_id;
    __u32 flags;
#define I915_CONTEXT_CREATE_FLAGS_USE_EXTENSIONS      (1u << 0);
#define I915_CONTEXT_CREATE_FLAGS_SINGLE_TIMELINE   (1u << 1);
#define I915_CONTEXT_CREATE_FLAGS_UNKNOWN           ( - (I915_CONTEXT_CREATE_FLAGS_
↪ SINGLE_TIMELINE << 1));
    __u64 extensions;
#define I915_CONTEXT_CREATE_EXT_SETPARAM 0;
#define I915_CONTEXT_CREATE_EXT_CLONE 1;
};
```

Members

ctx_id

Id of the created context (output)

flags

Supported flags are:

I915_CONTEXT_CREATE_FLAGS_USE_EXTENSIONS:

Extensions may be appended to this structure and driver must check for those. See **extensions**.

I915_CONTEXT_CREATE_FLAGS_SINGLE_TIMELINE

Created context will have single timeline.

extensions

Zero-terminated chain of extensions.

I915_CONTEXT_CREATE_EXT_SETPARAM: Context parameter to set or query during context creation. See [struct `drm_i915_gem_context_create_ext_setparam`](#).

I915_CONTEXT_CREATE_EXT_CLONE: This extension has been removed. On the off chance someone somewhere has attempted to use it, never re-use this extension number.

struct **drm_i915_gem_context_param**

Context parameter to set or query.

Definition:

```
struct drm_i915_gem_context_param {
    __u32 ctx_id;
    __u32 size;
    __u64 param;
#define I915_CONTEXT_PARAM_BAN_PERIOD      0x1;
#define I915_CONTEXT_PARAM_NO_ZEROMAP    0x2;
#define I915_CONTEXT_PARAM_GTT_SIZE      0x3;
#define I915_CONTEXT_PARAM_NO_ERROR_CAPTURE    0x4;
#define I915_CONTEXT_PARAM_BANNABLE      0x5;
#define I915_CONTEXT_PARAM_PRIORITY      0x6;
#define I915_CONTEXT_MAX_USER_PRIORITY    1023 ;
#define I915_CONTEXT_DEFAULT_PRIORITY    0;
#define I915_CONTEXT_MIN_USER_PRIORITY    -1023 ;
#define I915_CONTEXT_PARAM_SSEU          0x7;
#define I915_CONTEXT_PARAM_RECOVERABLE    0x8;
#define I915_CONTEXT_PARAM_VM            0x9;
#define I915_CONTEXT_PARAM_ENGINES        0xa;
#define I915_CONTEXT_PARAM_PERSISTENCE    0xb;
#define I915_CONTEXT_PARAM_RING_SIZE      0xc;
#define I915_CONTEXT_PARAM_PROTECTED_CONTENT 0xd;
    __u64 value;
};
```

Members**ctx_id**

Context id

size

Size of the parameter **value**

param

Parameter to set or query

value

Context parameter value to be set or queried

Virtual Engine uAPI

Virtual engine is a concept where userspace is able to configure a set of physical engines, submit a batch buffer, and let the driver execute it on any engine from the set as it sees fit.

This is primarily useful on parts which have multiple instances of a same class engine, like for example GT3+ Skylake parts with their two VCS engines.

For instance userspace can enumerate all engines of a certain class using the previously described *Engine Discovery uAPI*. After that userspace can create a GEM context with a placeholder slot for the virtual engine (using *I915_ENGINE_CLASS_INVALID* and *I915_ENGINE_CLASS_INVALID_NONE* for class and instance respectively) and finally using the *I915_CONTEXT_ENGINES_EXT_LOAD_BALANCE* extension place a virtual engine in the same reserved slot.

Example of creating a virtual engine and submitting a batch buffer to it:

```
I915_DEFINE_CONTEXT_ENGINES_LOAD_BALANCE(virtual, 2) = {
    .base.name = I915_CONTEXT_ENGINES_EXT_LOAD_BALANCE,
    .engine_index = 0, // Place this virtual engine into engine map slot 0
    .num_siblings = 2,
    .engines = { { I915_ENGINE_CLASS_VIDEO, 0 },
                  { I915_ENGINE_CLASS_VIDEO, 1 } },
};
I915_DEFINE_CONTEXT_PARAM_ENGINES(engines, 1) = {
    .engines = { { I915_ENGINE_CLASS_INVALID,
                  I915_ENGINE_CLASS_INVALID_NONE } },
    .extensions = to_user_pointer(&virtual), // Chains after load_balance_
    ↪extension
};
struct drm_i915_gem_context_create_ext_setparam p_engines = {
    .base = {
        .name = I915_CONTEXT_CREATE_EXT_SETPARAM,
    },
    .param = {
        .param = I915_CONTEXT_PARAM_ENGINES,
        .value = to_user_pointer(&engines),
        .size = sizeof(engines),
    },
};
struct drm_i915_gem_context_create_ext create = {
    .flags = I915_CONTEXT_CREATE_FLAGS_USE_EXTENSIONS,
    .extensions = to_user_pointer(&p_engines);
};

ctx_id = gem_context_create_ext(drm_fd, &create);
```

```
// Now we have created a GEM context with its engine map containing a
// single virtual engine. Submissions to this slot can go either to
// vcs0 or vcs1, depending on the load balancing algorithm used inside
// the driver. The load balancing is dynamic from one batch buffer to
// another and transparent to userspace.
```

```
...
execbuf.rsvid1 = ctx_id;
execbuf.flags = 0; // Submits to index 0 which is the virtual engine
gem_execbuf(drm_fd, &execbuf);
```

struct **i915_context_engines_parallel_submit**

Configure engine for parallel submission.

Definition:

```
struct i915_context_engines_parallel_submit {
    struct i915_user_extension base;
    __u16 engine_index;
    __u16 width;
    __u16 num_siblings;
    __u16 mbz16;
    __u64 flags;
    __u64 mbz64[3];
    struct i915_engine_class_instance engines[];
};
```

Members

base

base user extension.

engine_index

slot for parallel engine

width

number of contexts per parallel engine or in other words the number of batches in each submission

num_siblings

number of siblings per context or in other words the number of possible placements for each submission

mbz16

reserved for future use; must be zero

flags

all undefined flags must be zero, currently not defined flags

mbz64

reserved for future use; must be zero

engines

2-d array of engine instances to configure parallel engine

$\text{length} = \text{width}(i) * \text{num_siblings}(j)$ $\text{index} = j + i * \text{num_siblings}$

Description

Setup a slot in the context engine map to allow multiple BBs to be submitted in a single execbuf IOCTL. Those BBs will then be scheduled to run on the GPU in parallel. Multiple hardware contexts are created internally in the i915 to run these BBs. Once a slot is configured for N BBs only N BBs can be submitted in each execbuf IOCTL and this is implicit behavior e.g. The user doesn't tell the execbuf IOCTL there are N BBs, the execbuf IOCTL knows how many BBs there are based on the slot's configuration. The N BBs are the last N buffer objects or first N if I915_EXEC_BATCH_FIRST is set.

The default placement behavior is to create implicit bonds between each context if each context maps to more than 1 physical engine (e.g. context is a virtual engine). Also we only allow contexts of same engine class and these contexts must be in logically contiguous order. Examples of the placement behavior are described below. Lastly, the default is to not allow BBs to be preempted mid-batch. Rather insert coordinated preemption points on all hardware contexts between each set of BBs. Flags could be added in the future to change both of these default behaviors.

Returns -EINVAL if hardware context placement configuration is invalid or if the placement configuration isn't supported on the platform / submission interface. Returns -ENODEV if extension isn't supported on the platform / submission interface.

Examples syntax:

CS[X] = generic engine of same class, logical instance X
INVALID = I915_ENGINE_CLASS_INVALID, I915_ENGINE_CLASS_INVALID_NONE

Example 1 pseudo code:

```
set_engines(INVALID)
set_parallel(engine_index=0, width=2, num_siblings=1,
             engines=CS[0],CS[1])
```

Results in the following valid placement:

CS[0], CS[1]

Example 2 pseudo code:

```
set_engines(INVALID)
set_parallel(engine_index=0, width=2, num_siblings=2,
             engines=CS[0],CS[2],CS[1],CS[3])
```

Results in the following valid placements:

CS[0], CS[1]

CS[2], CS[3]

This can be thought of as two virtual engines, each containing two engines thereby making a 2D array. However, there are bonds tying the entries together and placing restrictions on how they can be scheduled. Specifically, the scheduler can choose only vertical columns from the 2D array. That is, CS[0] is bonded to CS[1] and CS[2] to CS[3]. So if the scheduler wants to submit to CS[0], it must also choose CS[1] and vice versa. Same for CS[2] requires also using CS[3].

VE[0] = CS[0], CS[2]

```
VE[1] = CS[1], CS[3]
```

Example 3 pseudo code:

```
set_engines(INVALID)
set_parallel(engine_index=0, width=2, num_siblings=2,
             engines=CS[0],CS[1],CS[1],CS[3])
```

Results in the following valid and invalid placements:

```
CS[0], CS[1]
```

```
CS[1], CS[3] - Not logically contiguous, return -EINVAL
```

Context Engine Map uAPI

Context engine map is a new way of addressing engines when submitting batch- buffers, replacing the existing way of using identifiers like *I915_EXEC_BLT* inside the flags field of `struct drm_i915_gem_execbuffer2`.

To use it created GEM contexts need to be configured with a list of engines the user is intending to submit to. This is accomplished using the *I915_CONTEXT_PARAM_ENGINES* parameter and `struct i915_context_param_engines`.

For such contexts the *I915_EXEC_RING_MASK* field becomes an index into the configured map.

Example of creating such context and submitting against it:

```
I915_DEFINE_CONTEXT_PARAM_ENGINES(engines, 2) = {
    .engines = { { I915_ENGINE_CLASS_RENDER, 0 },
                 { I915_ENGINE_CLASS_COPY, 0 } }
};
struct drm_i915_gem_context_create_ext_setparam p_engines = {
    .base = {
        .name = I915_CONTEXT_CREATE_EXT_SETPARAM,
    },
    .param = {
        .param = I915_CONTEXT_PARAM_ENGINES,
        .value = to_user_pointer(&engines),
        .size = sizeof(engines),
    },
};
struct drm_i915_gem_context_create_ext create = {
    .flags = I915_CONTEXT_CREATE_FLAGS_USE_EXTENSIONS,
    .extensions = to_user_pointer(&p_engines);
};

ctx_id = gem_context_create_ext(drm_fd, &create);

// We have now created a GEM context with two engines in the map:
// Index 0 points to rcs0 while index 1 points to bcs0. Other engines
// will not be accessible from this context.

...
execbuf.rsvd1 = ctx_id;
execbuf.flags = 0; // Submits to index 0, which is rcs0 for this context
```

```
gem_execbuf(drm_fd, &execbuf);

...
execbuf.rsvd1 = ctx_id;
execbuf.flags = 1; // Submits to index 0, which is bcs0 for this context
gem_execbuf(drm_fd, &execbuf);
```

struct **drm_i915_gem_context_create_ext_setparam**

Context parameter to set or query during context creation.

Definition:

```
struct drm_i915_gem_context_create_ext_setparam {
    struct i915_user_extension base;
    struct drm_i915_gem_context_param param;
};
```

Members

base

Extension link. See *struct i915_user_extension*.

param

Context parameter to set or query. See *struct drm_i915_gem_context_param*.

struct **drm_i915_gem_vm_control**

Structure to create or destroy VM.

Definition:

```
struct drm_i915_gem_vm_control {
    __u64 extensions;
    __u32 flags;
    __u32 vm_id;
};
```

Members

extensions

Zero-terminated chain of extensions.

flags

reserved for future usage, currently MBZ

vm_id

Id of the VM created or to be destroyed

Description

DRM_I915_GEM_VM_CREATE -

Create a new virtual memory address space (ppGTT) for use within a context on the same file. Extensions can be provided to configure exactly how the address space is setup upon creation.

The id of new VM (bound to the fd) for use with I915_CONTEXT_PARAM_VM is returned in the outparam **id**.

An extension chain maybe provided, starting with **extensions**, and terminated by the **next_extension** being 0. Currently, no extensions are defined.

DRM_I915_GEM_VM_DESTROY -

Destroys a previously created VM id, specified in **vm_id**.

No extensions or flags are allowed currently, and so must be zero.

struct **drm_i915_gem_userptr**

Create GEM object from user allocated memory.

Definition:

```
struct drm_i915_gem_userptr {
    __u64 user_ptr;
    __u64 user_size;
    __u32 flags;
#define I915_USERPTR_READ_ONLY 0x1;
#define I915_USERPTR_PROBE 0x2;
#define I915_USERPTR_UNSYNCHRONIZED 0x80000000;
    __u32 handle;
};
```

Members

user_ptr

The pointer to the allocated memory.

Needs to be aligned to PAGE_SIZE.

user_size

The size in bytes for the allocated memory. This will also become the object size.

Needs to be aligned to PAGE_SIZE, and should be at least PAGE_SIZE, or larger.

flags

Supported flags:

I915_USERPTR_READ_ONLY:

Mark the object as readonly, this also means GPU access can only be readonly. This is only supported on HW which supports readonly access through the GTT. If the HW can't support readonly access, an error is returned.

I915_USERPTR_PROBE:

Probe the provided **user_ptr** range and validate that the **user_ptr** is indeed pointing to normal memory and that the range is also valid. For example if some garbage address is given to the kernel, then this should complain.

Returns -EFAULT if the probe failed.

Note that this doesn't populate the backing pages, and also doesn't guarantee that the object will remain valid when the object is eventually used.

The kernel supports this feature if I915_PARAM_HAS_USERPTR_PROBE returns a non-zero value.

I915_USERPTR_UNSYNCHRONIZED:

NOT USED. Setting this flag will result in an error.

handle

Returned handle for the object.

Object handles are nonzero.

Description

Userptr objects have several restrictions on what ioctls can be used with the object handle.

struct **drm_i915_perf_oa_config**

Definition:

```
struct drm_i915_perf_oa_config {
    char uuid[36];
    __u32 n_mux_regs;
    __u32 n_boolean_regs;
    __u32 n_flex_regs;
    __u64 mux_regs_ptr;
    __u64 boolean_regs_ptr;
    __u64 flex_regs_ptr;
};
```

Members

uuid

String formatted like “%08x-%04x-%04x-%04x-%012x”

n_mux_regs

Number of mux regs in mux_regs_ptr.

n_boolean_regs

Number of boolean regs in boolean_regs_ptr.

n_flex_regs

Number of flex regs in flex_regs_ptr.

mux_regs_ptr

Pointer to tuples of u32 values (register address, value) for mux registers. Expected length of buffer is (2 * sizeof(u32) * n_mux_regs).

boolean_regs_ptr

Pointer to tuples of u32 values (register address, value) for mux registers. Expected length of buffer is (2 * sizeof(u32) * n_boolean_regs).

flex_regs_ptr

Pointer to tuples of u32 values (register address, value) for mux registers. Expected length of buffer is (2 * sizeof(u32) * n_flex_regs).

Description

Structure to upload perf dynamic configuration into the kernel.

struct **drm_i915_query_item**

An individual query for the kernel to process.

Definition:


```

struct drm_i915_query_item {
    __u64 query_id;
#define DRM_I915_QUERY_TOPOLOGY_INFO          1;
#define DRM_I915_QUERY_ENGINE_INFO           2;
#define DRM_I915_QUERY_PERF_CONFIG           3;
#define DRM_I915_QUERY_MEMORY_REGIONS       4;
#define DRM_I915_QUERY_HWCONFIG_BLOB        5;
#define DRM_I915_QUERY_GEOMETRY_SUBSLICES   6;
    __s32 length;
    __u32 flags;
#define DRM_I915_QUERY_PERF_CONFIG_LIST      1;
#define DRM_I915_QUERY_PERF_CONFIG_DATA_FOR_UUID 2;
#define DRM_I915_QUERY_PERF_CONFIG_DATA_FOR_ID 3;
    __u64 data_ptr;
};

```

Members

query_id

The id for this query. Currently accepted query IDs are:

- DRM_I915_QUERY_TOPOLOGY_INFO (see [struct drm_i915_query_topology_info](#))
- DRM_I915_QUERY_ENGINE_INFO (see [struct drm_i915_engine_info](#))
- DRM_I915_QUERY_PERF_CONFIG (see [struct drm_i915_query_perf_config](#))
- DRM_I915_QUERY_MEMORY_REGIONS (see [struct drm_i915_query_memory_regions](#))
- DRM_I915_QUERY_HWCONFIG_BLOB (see [GuC HWCONFIG blob uAPI](#))
- DRM_I915_QUERY_GEOMETRY_SUBSLICES (see [struct drm_i915_query_topology_info](#))

length

When set to zero by userspace, this is filled with the size of the data to be written at the **data_ptr** pointer. The kernel sets this value to a negative value to signal an error on a particular query item.

flags

When query_id == DRM_I915_QUERY_TOPOLOGY_INFO, must be 0.

When query_id == DRM_I915_QUERY_PERF_CONFIG, must be one of the following:

- DRM_I915_QUERY_PERF_CONFIG_LIST
- DRM_I915_QUERY_PERF_CONFIG_DATA_FOR_UUID
- DRM_I915_QUERY_PERF_CONFIG_FOR_UUID

When query_id == DRM_I915_QUERY_GEOMETRY_SUBSLICES must contain a [struct i915_engine_class_instance](#) that references a render engine.

data_ptr

Data will be written at the location pointed by **data_ptr** when the value of **length** matches the length of the data to be written by the kernel.

Description

The behaviour is determined by the **query_id**. Note that exactly what **data_ptr** is also depends on the specific **query_id**.

struct **drm_i915_query**

Supply an array of *struct drm_i915_query_item* for the kernel to fill out.

Definition:

```
struct drm_i915_query {
    __u32 num_items;
    __u32 flags;
    __u64 items_ptr;
};
```

Members

num_items

The number of elements in the **items_ptr** array

flags

Unused for now. Must be cleared to zero.

items_ptr

Pointer to an array of *struct drm_i915_query_item*. The number of array elements is **num_items**.

Description

Note that this is generally a two step process for each *struct drm_i915_query_item* in the array:

1. Call the `DRM_IOCTL_I915_QUERY`, giving it our array of *struct drm_i915_query_item*, with *drm_i915_query_item.length* set to zero. The kernel will then fill in the size, in bytes, which tells userspace how memory it needs to allocate for the blob(say for an array of properties).
2. Next we call `DRM_IOCTL_I915_QUERY` again, this time with the *drm_i915_query_item.data_ptr* equal to our newly allocated blob. Note that the *drm_i915_query_item.length* should still be the same as what the kernel previously set. At this point the kernel can fill in the blob.

Note that for some query items it can make sense for userspace to just pass in a buffer/blob equal to or larger than the required size. In this case only a single ioctl call is needed. For some smaller query items this can work quite well.

struct **drm_i915_query_topology_info**

Definition:

```
struct drm_i915_query_topology_info {
    __u16 flags;
    __u16 max_slices;
    __u16 max_subslices;
    __u16 max_eus_per_subslice;
    __u16 subslice_offset;
    __u16 subslice_stride;
    __u16 eu_offset;
```

```
__u16 eu_stride;
__u8 data[];
};
```

Members

flags

Unused for now. Must be cleared to zero.

max_slices

The number of bits used to express the slice mask.

max_subslices

The number of bits used to express the subslice mask.

max_eus_per_subslice

The number of bits in the EU mask that correspond to a single subslice's EUs.

subslice_offset

Offset in data[] at which the subslice masks are stored.

subslice_stride

Stride at which each of the subslice masks for each slice are stored.

eu_offset

Offset in data[] at which the EU masks are stored.

eu_stride

Stride at which each of the EU masks for each subslice are stored.

data

Contains 3 pieces of information :

- The slice mask with one bit per slice telling whether a slice is available. The availability of slice X can be queried with the following formula :

```
(data[X / 8] >> (X % 8)) & 1
```

Starting with Xe_HP platforms, Intel hardware no longer has traditional slices so i915 will always report a single slice (hardcoded slicemask = 0x1) which contains all of the platform's subslices. I.e., the mask here does not reflect any of the newer hardware concepts such as "gslices" or "cslices" since userspace is capable of inferring those from the subslice mask.

- The subslice mask for each slice with one bit per subslice telling whether a subslice is available. Starting with Gen12 we use the term "subslice" to refer to what the hardware documentation describes as a "dual-subslices." The availability of subslice Y in slice X can be queried with the following formula :

```
(data[subslice_offset + X * subslice_stride + Y / 8] >> (Y % 8)) & 1
```

- The EU mask for each subslice in each slice, with one bit per EU telling whether an EU is available. The availability of EU Z in subslice Y in slice X can be queried with the following formula :

```
(data[eu_offset +
      (X * max_subslices + Y) * eu_stride +
```

```

    Z / 8
] >> (Z % 8)) & 1

```

Description

Describes slice/subslice/EU information queried by `DRM_I915_QUERY_TOPOLOGY_INFO`

Engine Discovery uAPI

Engine discovery uAPI is a way of enumerating physical engines present in a GPU associated with an open i915 DRM file descriptor. This supersedes the old way of using `DRM_IOCTL_I915_GETPARAM` and engine identifiers like `I915_PARAM_HAS_BLT`.

The need for this interface came starting with Icelake and newer GPUs, which started to establish a pattern of having multiple engines of a same class, where not all instances were always completely functionally equivalent.

Entry point for this uapi is `DRM_IOCTL_I915_QUERY` with the `DRM_I915_QUERY_ENGINE_INFO` as the queried item id.

Example for getting the list of engines:

```

struct drm_i915_query_engine_info *info;
struct drm_i915_query_item item = {
    .query_id = DRM_I915_QUERY_ENGINE_INFO;
};
struct drm_i915_query query = {
    .num_items = 1,
    .items_ptr = (uintptr_t)&item,
};
int err, i;

// First query the size of the blob we need, this needs to be large
// enough to hold our array of engines. The kernel will fill out the
// item.length for us, which is the number of bytes we need.
//
// Alternatively a large buffer can be allocated straight away enabling
// querying in one pass, in which case item.length should contain the
// length of the provided buffer.
err = ioctl(fd, DRM_IOCTL_I915_QUERY, &query);
if (err) ...

info = calloc(1, item.length);
// Now that we allocated the required number of bytes, we call the ioctl
// again, this time with the data_ptr pointing to our newly allocated
// blob, which the kernel can then populate with info on all engines.
item.data_ptr = (uintptr_t)&info,

err = ioctl(fd, DRM_IOCTL_I915_QUERY, &query);
if (err) ...

// We can now access each engine in the array
for (i = 0; i < info->num_engines; i++) {
    struct drm_i915_engine_info einfo = info->engines[i];

```

```

        u16 class = einfo.engine.class;
        u16 instance = einfo.engine.instance;
        ....
    }

    free(info);

```

Each of the enumerated engines, apart from being defined by its class and instance (see [struct i915_engine_class_instance](#)), also can have flags and capabilities defined as documented in `i915_drm.h`.

For instance video engines which support HEVC encoding will have the `I915_VIDEO_CLASS_CAPABILITY_HEVC` capability bit set.

Engine discovery only fully comes to its own when combined with the new way of addressing engines when submitting batch buffers using contexts with engine maps configured.

struct **drm_i915_engine_info**

Definition:

```

struct drm_i915_engine_info {
    struct i915_engine_class_instance engine;
    __u32 rsvd0;
    __u64 flags;
#define I915_ENGINE_INFO_HAS_LOGICAL_INSTANCE        (1 << 0);
    __u64 capabilities;
#define I915_VIDEO_CLASS_CAPABILITY_HEVC            (1 << 0);
#define I915_VIDEO_AND_ENHANCE_CLASS_CAPABILITY_SFC (1 << 1);
    __u16 logical_instance;
    __u16 rsvd1[3];
    __u64 rsvd2[3];
};

```

Members

engine

Engine class and instance.

rsvd0

Reserved field.

flags

Engine flags.

capabilities

Capabilities of this engine.

logical_instance

Logical instance of engine

rsvd1

Reserved fields.

rsvd2

Reserved fields.

Description

Describes one engine and it's capabilities as known to the driver.

struct **drm_i915_query_engine_info**

Definition:

```
struct drm_i915_query_engine_info {
    __u32 num_engines;
    __u32 rsvd[3];
    struct drm_i915_engine_info engines[];
};
```

Members

num_engines

Number of *struct drm_i915_engine_info* structs following.

rsvd

MBZ

engines

Marker for *drm_i915_engine_info* structures.

Description

Engine info query enumerates all engines known to the driver by filling in an array of *struct drm_i915_engine_info* structures.

struct **drm_i915_query_perf_config**

Definition:

```
struct drm_i915_query_perf_config {
    union {
        __u64 n_configs;
        __u64 config;
        char uuid[36];
    };
    __u32 flags;
    __u8 data[];
};
```

Members

{unnamed_union}

anonymous

n_configs

When *drm_i915_query_item.flags* == *DRM_I915_QUERY_PERF_CONFIG_LIST*, i915 sets this fields to the number of configurations available.

config

When *drm_i915_query_item.flags* == *DRM_I915_QUERY_PERF_CONFIG_DATA_FOR_ID*, i915 will use the value in this field as configuration identifier to decide what data to write into *config_ptr*.

uuid

When `drm_i915_query_item.flags == DRM_I915_QUERY_PERF_CONFIG_DATA_FOR_UUID`, i915 will use the value in this field as configuration identifier to decide what data to write into `config_ptr`.

String formatted like "08x-````04x-````04x-````04x-````012x"

flags

Unused for now. Must be cleared to zero.

data

When `drm_i915_query_item.flags == DRM_I915_QUERY_PERF_CONFIG_LIST`, i915 will write an array of `__u64` of configuration identifiers.

When `drm_i915_query_item.flags == DRM_I915_QUERY_PERF_CONFIG_DATA`, i915 will write a `struct drm_i915_perf_oa_config`. If the following fields of `struct drm_i915_perf_oa_config` are not set to 0, i915 will write into the associated pointers the values of submitted when the configuration was created :

- `drm_i915_perf_oa_config.n_mux_regs`
- `drm_i915_perf_oa_config.n_boolean_regs`
- `drm_i915_perf_oa_config.n_flex_regs`

Description

Data written by the kernel with query `DRM_I915_QUERY_PERF_CONFIG` and `DRM_I915_QUERY_GEOMETRY_SUBSLICES`.

enum `drm_i915_gem_memory_class`

Supported memory classes

Constants**`I915_MEMORY_CLASS_SYSTEM`**

System memory

`I915_MEMORY_CLASS_DEVICE`

Device local-memory

struct `drm_i915_gem_memory_class_instance`

Identify particular memory region

Definition:

```
struct drm_i915_gem_memory_class_instance {
    __u16 memory_class;
    __u16 memory_instance;
};
```

Members**`memory_class`**

See `enum drm_i915_gem_memory_class`

`memory_instance`

Which instance

struct **drm_i915_memory_region_info**

Describes one region as known to the driver.

Definition:

```
struct drm_i915_memory_region_info {
    struct drm_i915_gem_memory_class_instance region;
    __u32 rsvd0;
    __u64 probed_size;
    __u64 unallocated_size;
    union {
        __u64 rsvd1[8];
        struct {
            __u64 probed_cpu_visible_size;
            __u64 unallocated_cpu_visible_size;
        };
    };
};
```

Members

region

The class:instance pair encoding

rsvd0

MBZ

probed_size

Memory probed by the driver

Note that it should not be possible to ever encounter a zero value here, also note that no current region type will ever return -1 here. Although for future region types, this might be a possibility. The same applies to the other size fields.

unallocated_size

Estimate of memory remaining

Requires CAP_PERFMON or CAP_SYS_ADMIN to get reliable accounting. Without this (or if this is an older kernel) the value here will always equal the **probed_size**. Note this is only currently tracked for I915_MEMORY_CLASS_DEVICE regions (for other types the value here will always equal the **probed_size**).

{unnamed_union}

anonymous

rsvd1

MBZ

{unnamed_struct}

anonymous

probed_cpu_visible_size

Memory probed by the driver that is CPU accessible.

This will be always be \leq **probed_size**, and the remainder (if there is any) will not be CPU accessible.

On systems without small BAR, the **probed_size** will always equal the **probed_cpu_visible_size**, since all of it will be CPU accessible.

Note this is only tracked for I915_MEMORY_CLASS_DEVICE regions (for other types the value here will always equal the **probed_size**).

Note that if the value returned here is zero, then this must be an old kernel which lacks the relevant small-bar uAPI support (including I915_GEM_CREATE_EXT_FLAG_NEEDS_CPU_ACCESS), but on such systems we should never actually end up with a small BAR configuration, assuming we are able to load the kernel module. Hence it should be safe to treat this the same as when **probed_cpu_visible_size == probed_size**.

unallocated_cpu_visible_size

Estimate of CPU visible memory remaining.

Note this is only tracked for I915_MEMORY_CLASS_DEVICE regions (for other types the value here will always equal the **probed_cpu_visible_size**).

Requires CAP_PERFMON or CAP_SYS_ADMIN to get reliable accounting. Without this the value here will always equal the **probed_cpu_visible_size**. Note this is only currently tracked for I915_MEMORY_CLASS_DEVICE regions (for other types the value here will also always equal the **probed_cpu_visible_size**).

If this is an older kernel the value here will be zero, see also **probed_cpu_visible_size**.

Description

Note this is using both *struct drm_i915_query_item* and *struct drm_i915_query*. For this new query we are adding the new query id DRM_I915_QUERY_MEMORY_REGIONS at *drm_i915_query_item.query_id*.

struct **drm_i915_query_memory_regions**

Definition:

```
struct drm_i915_query_memory_regions {
    __u32 num_regions;
    __u32 rsvd[3];
    struct drm_i915_memory_region_info regions[];
};
```

Members

num_regions

Number of supported regions

rsvd

MBZ

regions

Info about each supported region

Description

The region info query enumerates all regions known to the driver by filling in an array of *struct drm_i915_memory_region_info* structures.

Example for getting the list of supported regions:

```
struct drm_i915_query_memory_regions *info;
struct drm_i915_query_item item = {
    .query_id = DRM_I915_QUERY_MEMORY_REGIONS;
};
struct drm_i915_query query = {
    .num_items = 1,
    .items_ptr = (uintptr_t)&item,
};
int err, i;

// First query the size of the blob we need, this needs to be large
// enough to hold our array of regions. The kernel will fill out the
// item.length for us, which is the number of bytes we need.
err = ioctl(fd, DRM_IOCTL_I915_QUERY, &query);
if (err) ...

info = calloc(1, item.length);
// Now that we allocated the required number of bytes, we call the ioctl
// again, this time with the data_ptr pointing to our newly allocated
// blob, which the kernel can then populate with the all the region info.
item.data_ptr = (uintptr_t)&info,

err = ioctl(fd, DRM_IOCTL_I915_QUERY, &query);
if (err) ...

// We can now access each region in the array
for (i = 0; i < info->num_regions; i++) {
    struct drm_i915_memory_region_info mr = info->regions[i];
    u16 class = mr.region.class;
    u16 instance = mr.region.instance;

    ....
}

free(info);
```

GuC HWCONFIG blob uAPI

The GuC produces a blob with information about the current device. i915 reads this blob from GuC and makes it available via this uAPI.

The format and meaning of the blob content are documented in the Programmer's Reference Manual.

struct **drm_i915_gem_create_ext**

Existing `gem_create` behaviour, with added extension support using *struct i915_user_extension*.

Definition:

```
struct drm_i915_gem_create_ext {
    __u64 size;
    __u32 handle;
```

```
#define I915_GEM_CREATE_EXT_FLAG_NEEDS_CPU_ACCESS (1 << 0);
    __u32 flags;
#define I915_GEM_CREATE_EXT_MEMORY_REGIONS 0;
#define I915_GEM_CREATE_EXT_PROTECTED_CONTENT 1;
#define I915_GEM_CREATE_EXT_SET_PAT 2;
    __u64 extensions;
};
```

Members

size

Requested size for the object.

The (page-aligned) allocated size for the object will be returned.

On platforms like DG2/ATS the kernel will always use 64K or larger pages for I915_MEMORY_CLASS_DEVICE. The kernel also requires a minimum of 64K GTT alignment for such objects.

NOTE: Previously the ABI here required a minimum GTT alignment of 2M on DG2/ATS, due to how the hardware implemented 64K GTT page support, where we had the following complications:

- 1) The entire PDE (which covers a 2MB virtual address range), must contain only 64K PTEs, i.e mixing 4K and 64K PTEs in the same PDE is forbidden by the hardware.
- 2) We still need to support 4K PTEs for I915_MEMORY_CLASS_SYSTEM objects.

However on actual production HW this was completely changed to now allow setting a TLB hint at the PTE level (see PS64), which is a lot more flexible than the above. With this the 2M restriction was dropped where we now only require 64K.

handle

Returned handle for the object.

Object handles are nonzero.

flags

Optional flags.

Supported values:

I915_GEM_CREATE_EXT_FLAG_NEEDS_CPU_ACCESS - Signal to the kernel that the object will need to be accessed via the CPU.

Only valid when placing objects in I915_MEMORY_CLASS_DEVICE, and only strictly required on configurations where some subset of the device memory is directly visible/mappable through the CPU (which we also call small BAR), like on some DG2+ systems. Note that this is quite undesirable, but due to various factors like the client CPU, BIOS etc it's something we can expect to see in the wild. See [drm_i915_memory_region_info.probed_cpu_visible_size](#) for how to determine if this system applies.

Note that one of the placements MUST be I915_MEMORY_CLASS_SYSTEM, to ensure the kernel can always spill the allocation to system memory, if the object can't be allocated in the mappable part of I915_MEMORY_CLASS_DEVICE.

Also note that since the kernel only supports flat-CCS on objects that can *only* be placed in `I915_MEMORY_CLASS_DEVICE`, we therefore don't support `I915_GEM_CREATE_EXT_FLAG_NEEDS_CPU_ACCESS` together with flat-CCS.

Without this hint, the kernel will assume that non-mappable `I915_MEMORY_CLASS_DEVICE` is preferred for this object. Note that the kernel can still migrate the object to the mappable part, as a last resort, if userspace ever CPU faults this object, but this might be expensive, and so ideally should be avoided.

On older kernels which lack the relevant small-bar uAPI support (see also [`drm_i915_memory_region_info.probed_cpu_visible_size`](#)), usage of the flag will result in an error, but it should NEVER be possible to end up with a small BAR configuration, assuming we can also successfully load the i915 kernel module. In such cases the entire `I915_MEMORY_CLASS_DEVICE` region will be CPU accessible, and as such there are zero restrictions on where the object can be placed.

extensions

The chain of extensions to apply to this object.

This will be useful in the future when we need to support several different extensions, and we need to apply more than one when creating the object. See [`struct i915_user_extension`](#).

If we don't supply any extensions then we get the same old `gem_create` behaviour.

For `I915_GEM_CREATE_EXT_MEMORY_REGIONS` usage see [`struct drm_i915_gem_create_ext_memory_regions`](#).

For `I915_GEM_CREATE_EXT_PROTECTED_CONTENT` usage see [`struct drm_i915_gem_create_ext_protected_content`](#).

For `I915_GEM_CREATE_EXT_SET_PAT` usage see [`struct drm_i915_gem_create_ext_set_pat`](#).

Description

Note that new buffer flags should be added here, at least for the stuff that is immutable. Previously we would have two ioctls, one to create the object with `gem_create`, and another to apply various parameters, however this creates some ambiguity for the `params` which are considered immutable. Also in general we're phasing out the various SET/GET ioctls.

struct **drm_i915_gem_create_ext_memory_regions**

The `I915_GEM_CREATE_EXT_MEMORY_REGIONS` extension.

Definition:

```
struct drm_i915_gem_create_ext_memory_regions {
    struct i915_user_extension base;
    __u32 pad;
    __u32 num_regions;
    __u64 regions;
};
```

Members

base

Extension link. See [`struct i915_user_extension`](#).

pad

MBZ

num_regionsNumber of elements in the **regions** array.**regions**

The regions/placements array.

An array of *struct drm_i915_gem_memory_class_instance*.**Description**

Set the object with the desired set of placements/regions in priority order. Each entry must be unique and supported by the device.

This is provided as an array of *struct drm_i915_gem_memory_class_instance*, or an equivalent layout of class:instance pair encodings. See *struct drm_i915_query_memory_regions* and `DRM_I915_QUERY_MEMORY_REGIONS` for how to query the supported regions for a device.

As an example, on discrete devices, if we wish to set the placement as device local-memory we can do something like:

```
struct drm_i915_gem_memory_class_instance region_lmem = {
    .memory_class = I915_MEMORY_CLASS_DEVICE,
    .memory_instance = 0,
};
struct drm_i915_gem_create_ext_memory_regions regions = {
    .base = { .name = I915_GEM_CREATE_EXT_MEMORY_REGIONS },
    .regions = (uintptr_t)&region_lmem,
    .num_regions = 1,
};
struct drm_i915_gem_create_ext create_ext = {
    .size = 16 * PAGE_SIZE,
    .extensions = (uintptr_t)&regions,
};

int err = ioctl(fd, DRM_IOCTL_I915_GEM_CREATE_EXT, &create_ext);
if (err) ...
```

At which point we get the object handle in *drm_i915_gem_create_ext.handle*, along with the final object size in *drm_i915_gem_create_ext.size*, which should account for any rounding up, if required.

Note that userspace has no means of knowing the current backing region for objects where **num_regions** is larger than one. The kernel will only ensure that the priority order of the **regions** array is honoured, either when initially placing the object, or when moving memory around due to memory pressure

On Flat-CCS capable HW, compression is supported for the objects residing in `I915_MEMORY_CLASS_DEVICE`. When such objects (compressed) have other memory class in **regions** and migrated (by i915, due to memory constraints) to the non `I915_MEMORY_CLASS_DEVICE` region, then i915 needs to decompress the content. But i915 doesn't have the required information to decompress the userspace compressed objects.

So i915 supports Flat-CCS, on the objects which can reside only on I915_MEMORY_CLASS_DEVICE regions.

struct **drm_i915_gem_create_ext_protected_content**

The I915_OBJECT_PARAM_PROTECTED_CONTENT extension.

Definition:

```
struct drm_i915_gem_create_ext_protected_content {
    struct i915_user_extension base;
    __u32 flags;
};
```

Members

base

Extension link. See [struct i915_user_extension](#).

flags

reserved for future usage, currently MBZ

Description

If this extension is provided, buffer contents are expected to be protected by PXP encryption and require decryption for scan out and processing. This is only possible on platforms that have PXP enabled, on all other scenarios using this extension will cause the ioctl to fail and return -ENODEV. The flags parameter is reserved for future expansion and must currently be set to zero.

The buffer contents are considered invalid after a PXP session teardown.

The encryption is guaranteed to be processed correctly only if the object is submitted with a context created using the I915_CONTEXT_PARAM_PROTECTED_CONTENT flag. This will also enable extra checks at submission time on the validity of the objects involved.

Below is an example on how to create a protected object:

```
struct drm_i915_gem_create_ext_protected_content protected_ext = {
    .base = { .name = I915_GEM_CREATE_EXT_PROTECTED_CONTENT },
    .flags = 0,
};
struct drm_i915_gem_create_ext create_ext = {
    .size = PAGE_SIZE,
    .extensions = (uintptr_t)&protected_ext,
};

int err = ioctl(fd, DRM_IOCTL_I915_GEM_CREATE_EXT, &create_ext);
if (err) ...
```

struct **drm_i915_gem_create_ext_set_pat**

The I915_GEM_CREATE_EXT_SET_PAT extension.

Definition:

```
struct drm_i915_gem_create_ext_set_pat {
    struct i915_user_extension base;
```

```

    __u32 pat_index;
    __u32 rsvd;
};

```

Members

base

Extension link. See *struct i915_user_extension*.

pat_index

PAT index to be set PAT index is a bit field in Page Table Entry to control caching behaviors for GPU accesses. The definition of PAT index is platform dependent and can be found in hardware specifications,

rsvd

reserved for future use

Description

If this extension is provided, the specified caching policy (PAT index) is applied to the buffer object.

Below is an example on how to create an object with specific caching policy:

```

struct drm_i915_gem_create_ext_set_pat set_pat_ext = {
    .base = { .name = I915_GEM_CREATE_EXT_SET_PAT },
    .pat_index = 0,
};
struct drm_i915_gem_create_ext create_ext = {
    .size = PAGE_SIZE,
    .extensions = (uintptr_t)&set_pat_ext,
};

int err = ioctl(fd, DRM_IOCTL_I915_GEM_CREATE_EXT, &create_ext);
if (err) ...

```

8.2 drm/nouveau uAPI

8.2.1 VM_BIND / EXEC uAPI

Nouveau's VM_BIND / EXEC UAPI consists of three ioctls: DRM_NOUVEAU_VM_INIT, DRM_NOUVEAU_VM_BIND and DRM_NOUVEAU_EXEC.

In order to use the UAPI firstly a user client must initialize the VA space using the DRM_NOUVEAU_VM_INIT ioctl specifying which region of the VA space should be managed by the kernel and which by the UMD.

The DRM_NOUVEAU_VM_BIND ioctl provides clients an interface to manage the userspace-managable portion of the VA space. It provides operations to map and unmap memory. Mappings may be flagged as sparse. Sparse mappings are not backed by a GEM object and the kernel will ignore GEM handles provided alongside a sparse mapping.

Userspace may request memory backed mappings either within or outside of the bounds (but not crossing those bounds) of a previously mapped sparse mapping. Subsequently requested memory backed mappings within a sparse mapping will take precedence over the corresponding range of the sparse mapping. If such memory backed mappings are unmapped the kernel will make sure that the corresponding sparse mapping will take their place again. Requests to unmap a sparse mapping that still contains memory backed mappings will result in those memory backed mappings being unmapped first.

Unmap requests are not bound to the range of existing mappings and can even overlap the bounds of sparse mappings. For such a request the kernel will make sure to unmap all memory backed mappings within the given range, splitting up memory backed mappings which are only partially contained within the given range. Unmap requests with the sparse flag set must match the range of a previously mapped sparse mapping exactly though.

While the kernel generally permits arbitrary sequences and ranges of memory backed mappings being mapped and unmapped, either within a single or multiple `VM_BIND` ioctl calls, there are some restrictions for sparse mappings.

The kernel does not permit to:

- unmap non-existent sparse mappings
- unmap a sparse mapping and map a new sparse mapping overlapping the range of the previously unmapped sparse mapping within the same `VM_BIND` ioctl
- unmap a sparse mapping and map new memory backed mappings overlapping the range of the previously unmapped sparse mapping within the same `VM_BIND` ioctl

When using the `VM_BIND` ioctl to request the kernel to map memory to a given virtual address in the GPU's VA space there is no guarantee that the actual mappings are created in the GPU's MMU. If the given memory is swapped out at the time the bind operation is executed the kernel will stash the mapping details into it's internal allocator and create the actual MMU mappings once the memory is swapped back in. While this is transparent for userspace, it is guaranteed that all the backing memory is swapped back in and all the memory mappings, as requested by userspace previously, are actually mapped once the `DRM_NOUVEAU_EXEC` ioctl is called to submit an exec job.

A `VM_BIND` job can be executed either synchronously or asynchronously. If executed asynchronously, userspace may provide a list of syncobjs this job will wait for and/or a list of syncobj the kernel will signal once the `VM_BIND` job finished execution. If executed synchronously the ioctl will block until the bind job is finished. For synchronous jobs the kernel will not permit any syncobjs submitted to the kernel.

To execute a push buffer the UAPI provides the `DRM_NOUVEAU_EXEC` ioctl. EXEC jobs are always executed asynchronously, and, equal to `VM_BIND` jobs, provide the option to synchronize them with syncobjs.

Besides that, EXEC jobs can be scheduled for a specified channel to execute on.

Since `VM_BIND` jobs update the GPU's VA space on job submit, EXEC jobs do have an up to date view of the VA space. However, the actual mappings might still be pending. Hence, EXEC jobs require to have the particular fences - of the corresponding `VM_BIND` jobs they depend on - attached to them.

struct **drm_nouveau_sync**
 sync object

Definition:


```

struct drm_nouveau_sync {
    __u32 flags;
#define DRM_NOUVEAU_SYNC_SYNCOBJ 0x0;
#define DRM_NOUVEAU_SYNC_TIMELINE_SYNCOBJ 0x1;
#define DRM_NOUVEAU_SYNC_TYPE_MASK 0xf;
    __u32 handle;
    __u64 timeline_value;
};

```

Members

flags

the flags for a sync object

The first 8 bits are used to determine the type of the sync object.

handle

the handle of the sync object

timeline_value

The timeline point of the sync object in case the syncobj is of type `DRM_NOUVEAU_SYNC_TIMELINE_SYNCOBJ`.

Description

This structure serves as synchronization mechanism for (potentially) asynchronous operations such as `EXEC` or `VM_BIND`.

struct `drm_nouveau_vm_init`

GPU VA space init structure

Definition:

```

struct drm_nouveau_vm_init {
    __u64 kernel_managed_addr;
    __u64 kernel_managed_size;
};

```

Members

kernel_managed_addr

start address of the kernel managed VA space region

kernel_managed_size

size of the kernel managed VA space region in bytes

Description

Used to initialize the GPU's VA space for a user client, telling the kernel which portion of the VA space is managed by the UMD and kernel respectively.

For the UMD to use the `VM_BIND` uAPI, this must be called before any BOs or channels are created; if called afterwards `DRM_IOCTL_NOUVEAU_VM_INIT` fails with `-ENOSYS`.

struct `drm_nouveau_vm_bind_op`

`VM_BIND` operation

Definition:

```
struct drm_nouveau_vm_bind_op {
    __u32 op;
#define DRM_NOUVEAU_VM_BIND_OP_MAP 0x0;
#define DRM_NOUVEAU_VM_BIND_OP_UNMAP 0x1;
    __u32 flags;
#define DRM_NOUVEAU_VM_BIND_SPARSE (1 << 8);
    __u32 handle;
    __u32 pad;
    __u64 addr;
    __u64 bo_offset;
    __u64 range;
};
```

Members

- op**
the operation type
- flags**
the flags for a [drm_nouveau_vm_bind_op](#)
- handle**
the handle of the DRM GEM object to map
- pad**
32 bit padding, should be 0
- addr**
the address the VA space region or (memory backed) mapping should be mapped to
- bo_offset**
the offset within the BO backing the mapping
- range**
the size of the requested mapping in bytes

Description

This structure represents a single VM_BIND operation. UMDs should pass an array of this structure via [struct drm_nouveau_vm_bind](#)'s op_ptr field.

struct **drm_nouveau_vm_bind**
structure for DRM_IOCTL_NOUVEAU_VM_BIND

Definition:

```
struct drm_nouveau_vm_bind {
    __u32 op_count;
    __u32 flags;
#define DRM_NOUVEAU_VM_BIND_RUN_ASYNC 0x1;
    __u32 wait_count;
    __u32 sig_count;
    __u64 wait_ptr;
    __u64 sig_ptr;
    __u64 op_ptr;
};
```

Members

op_count

the number of *drm_nouveau_vm_bind_op*

flags

the flags for a *drm_nouveau_vm_bind* ioctl

wait_count

the number of wait *drm_nouveau_syncs*

sig_count

the number of *drm_nouveau_syncs* to signal when finished

wait_ptr

pointer to *drm_nouveau_syncs* to wait for

sig_ptr

pointer to *drm_nouveau_syncs* to signal when finished

op_ptr

pointer to the *drm_nouveau_vm_bind_ops* to execute

struct *drm_nouveau_exec_push*

EXEC push operation

Definition:

```

struct drm_nouveau_exec_push {
    __u64 va;
    __u32 va_len;
    __u32 flags;
#define DRM_NOUVEAU_EXEC_PUSH_NO_PREFETCH 0x1;
};

```

Members

va

the virtual address of the push buffer mapping

va_len

the length of the push buffer mapping

flags

the flags for this push buffer mapping

Description

This structure represents a single EXEC push operation. UMDs should pass an array of this structure via *struct drm_nouveau_exec*'s *push_ptr* field.

struct *drm_nouveau_exec*

structure for *DRM_IOCTL_NOUVEAU_EXEC*

Definition:

```

struct drm_nouveau_exec {
    __u32 channel;
    __u32 push_count;
};

```

```
__u32 wait_count;
__u32 sig_count;
__u64 wait_ptr;
__u64 sig_ptr;
__u64 push_ptr;
};
```

Members

channel

the channel to execute the push buffer in

push_count

the number of *drm_nouveau_exec_push* ops

wait_count

the number of wait *drm_nouveau_syncs*

sig_count

the number of *drm_nouveau_syncs* to signal when finished

wait_ptr

pointer to *drm_nouveau_syncs* to wait for

sig_ptr

pointer to *drm_nouveau_syncs* to signal when finished

push_ptr

pointer to *drm_nouveau_exec_push* ops

KERNEL CLIENTS

This library provides support for clients running in the kernel like fbdev and bootsplash. GEM drivers which provide a GEM based dumb buffer with a virtual address are supported.

struct **drm_client_funcs**
 DRM client callbacks

Definition:

```
struct drm_client_funcs {
    struct module *owner;
    void (*unregister)(struct drm_client_dev *client);
    int (*restore)(struct drm_client_dev *client);
    int (*hotplug)(struct drm_client_dev *client);
};
```

Members

owner

The module owner

unregister

Called when *drm_device* is unregistered. The client should respond by releasing its resources using *drm_client_release()*.

This callback is optional.

restore

Called on *drm_lastclose()*. The first client instance in the list that returns zero gets the privilege to restore and no more clients are called. This callback is not called after **unregister** has been called.

Note that the core does not guarantee exclusion against concurrent *drm_open()*. Clients need to ensure this themselves, for example by using *drm_master_internal_acquire()* and *drm_master_internal_release()*.

This callback is optional.

hotplug

Called on *drm_kms_helper_hotplug_event()*. This callback is not called after **unregister** has been called.

This callback is optional.

struct **drm_client_dev**
DRM client instance

Definition:

```
struct drm_client_dev {
    struct drm_device *dev;
    const char *name;
    struct list_head list;
    const struct drm_client_funcs *funcs;
    struct drm_file *file;
    struct mutex modeset_mutex;
    struct drm_mode_set *modesets;
    bool hotplug_failed;
};
```

Members

dev
DRM device

name
Name of the client.

list
List of all clients of a DRM device, linked into *drm_device.clientlist*. Protected by *drm_device.clientlist_mutex*.

funcs
DRM client functions (optional)

file
DRM file

modeset_mutex
Protects **modesets**.

modesets
CRTC configurations

hotplug_failed
Set by client hotplug helpers if the hotplugging failed before. It is usually not tried again.

struct **drm_client_buffer**
DRM client buffer

Definition:

```
struct drm_client_buffer {
    struct drm_client_dev *client;
    u32 pitch;
    struct drm_gem_object *gem;
    struct iosys_map map;
    struct drm_framebuffer *fb;
};
```

Members

client

DRM client

pitch

Buffer pitch

gem

GEM object backing this buffer

map

Virtual address for the buffer

fb

DRM framebuffer

drm_client_for_each_modeset

`drm_client_for_each_modeset (modeset, client)`

Iterate over client modesets

Parameters**modeset**

drm_mode_set loop cursor

client

DRM client

drm_client_for_each_connector_iter

`drm_client_for_each_connector_iter (connector, iter)`

connector_list iterator macro

Parameters**connector**

struct drm_connector pointer used as cursor

iter

struct drm_connector_list_iter

Description

This iterates the connectors that are useable for internal clients (excludes writeback connectors).

For more info see *drm_for_each_connector_iter()*.

`int drm_client_init(struct drm_device *dev, struct drm_client_dev *client, const char *name, const struct drm_client_funcs *funcs)`

Initialise a DRM client

Parameters

`struct drm_device *dev`

DRM device

`struct drm_client_dev *client`

DRM client

const char *name

Client name

const struct drm_client_funcs *funcs

DRM client functions (optional)

Description

This initialises the client and opens a *drm_file*. Use *drm_client_register()* to complete the process. The caller needs to hold a reference on **dev** before calling this function. The client is freed when the *drm_device* is unregistered. See *drm_client_release()*.

Return

Zero on success or negative error code on failure.

void **drm_client_register**(struct *drm_client_dev* *client)

Register client

Parameters

struct drm_client_dev *client

DRM client

Description

Add the client to the *drm_device* client list to activate its callbacks. **client** must be initialized by a call to *drm_client_init()*. After *drm_client_register()* it is no longer permissible to call *drm_client_release()* directly (outside the unregister callback), instead cleanup will happen automatically on driver unload.

Registering a client generates a hotplug event that allows the client to set up its display from pre-existing outputs. The client must have initialized its state to be able to handle the hotplug event successfully.

void **drm_client_release**(struct *drm_client_dev* *client)

Release DRM client resources

Parameters

struct drm_client_dev *client

DRM client

Description

Releases resources by closing the *drm_file* that was opened by *drm_client_init()*. It is called automatically if the *drm_client_funcs.unregister* callback is *_not_set*.

This function should only be called from the unregister callback. An exception is fbdev which cannot free the buffer if userspace has open file descriptors.

Note

Clients cannot initiate a release by themselves. This is done to keep the code simple. The driver has to be unloaded before the client can be unloaded.

void **drm_client_dev_hotplug**(struct *drm_device* *dev)

Send hotplug event to clients

Parameters

struct drm_device *dev
DRM device

Description

This function calls the `drm_client_funcs.hotplug` callback on the attached clients.

`drm_kms_helper_hotplug_event()` calls this function, so drivers that use it don't need to call this function themselves.

int **drm_client_buffer_vmap**(struct `drm_client_buffer` *buffer, struct iosys_map *map_copy)
Map DRM client buffer into address space

Parameters

struct drm_client_buffer *buffer
DRM client buffer

struct iosys_map *map_copy
Returns the mapped memory's address

Description

This function maps a client buffer into kernel address space. If the buffer is already mapped, it returns the existing mapping's address.

Client buffer mappings are not ref'counted. Each call to `drm_client_buffer_vmap()` should be followed by a call to `drm_client_buffer_vunmap()`; or the client buffer should be mapped throughout its lifetime.

The returned address is a copy of the internal value. In contrast to other vmap interfaces, you don't need it for the client's vunmap function. So you can modify it at will during blit and draw operations.

Return

0 on success, or a negative errno code otherwise.

void **drm_client_buffer_vunmap**(struct `drm_client_buffer` *buffer)
Unmap DRM client buffer

Parameters

struct drm_client_buffer *buffer
DRM client buffer

Description

This function removes a client buffer's memory mapping. Calling this function is only required by clients that manage their buffer mappings by themselves.

struct `drm_client_buffer` ***drm_client_framebuffer_create**(struct `drm_client_dev` *client,
u32 width, u32 height, u32
format)

Create a client framebuffer

Parameters

struct drm_client_dev *client
DRM client

u32 width

Framebuffer width

u32 height

Framebuffer height

u32 format

Buffer format

Description

This function creates a *drm_client_buffer* which consists of a *drm_framebuffer* backed by a dumb buffer. Call *drm_client_framebuffer_delete()* to free the buffer.

Return

Pointer to a client buffer or an error pointer on failure.

void **drm_client_framebuffer_delete**(struct *drm_client_buffer* *buffer)

Delete a client framebuffer

Parameters

struct drm_client_buffer *buffer

DRM client buffer (can be NULL)

int **drm_client_framebuffer_flush**(struct *drm_client_buffer* *buffer, struct *drm_rect* *rect)

Manually flush client framebuffer

Parameters

struct drm_client_buffer *buffer

DRM client buffer (can be NULL)

struct drm_rect *rect

Damage rectangle (if NULL flushes all)

Description

This calls *drm_framebuffer_funcs->dirty* (if present) to flush buffer changes for drivers that need it.

Return

Zero on success or negative error code on failure.

int **drm_client_modeset_probe**(struct *drm_client_dev* *client, unsigned int width, unsigned int height)

Probe for displays

Parameters

struct drm_client_dev *client

DRM client

unsigned int width

Maximum display mode width (optional)

unsigned int height

Maximum display mode height (optional)

Description

This function sets up display pipelines for enabled connectors and stores the config in the client's modeset array.

Return

Zero on success or negative error code on failure.

bool **drm_client_rotation**(struct *drm_mode_set* *modeset, unsigned int *rotation)
Check the initial rotation value

Parameters

struct drm_mode_set *modeset
DRM modeset

unsigned int *rotation
Returned rotation value

Description

This function checks if the primary plane in **modeset** can hw rotate to match the rotation needed on its connector.

Note

Currently only 0 and 180 degrees are supported.

Return

True if the plane can do the rotation, false otherwise.

int **drm_client_modeset_check**(struct *drm_client_dev* *client)
Check modeset configuration

Parameters

struct drm_client_dev *client
DRM client

Description

Check modeset configuration.

Return

Zero on success or negative error code on failure.

int **drm_client_modeset_commit_locked**(struct *drm_client_dev* *client)
Force commit CRTC configuration

Parameters

struct drm_client_dev *client
DRM client

Description

Commit modeset configuration to crtcs without checking if there is a DRM master. The assumption is that the caller already holds an internal DRM master reference acquired with `drm_master_internal_acquire()`.

Return

Zero on success or negative error code on failure.

int **drm_client_modeset_commit**(struct *drm_client_dev* *client)

Commit CRTC configuration

Parameters

struct *drm_client_dev* *client

DRM client

Description

Commit modeset configuration to crtcs.

Return

Zero on success or negative error code on failure.

int **drm_client_modeset_dpms**(struct *drm_client_dev* *client, int mode)

Set DPMS mode

Parameters

struct *drm_client_dev* *client

DRM client

int mode

DPMS mode

Note

For atomic drivers **mode** is reduced to on/off.

Return

Zero on success or negative error code on failure.

GPU DRIVER DOCUMENTATION

10.1 drm/amdgpu AMDgpu driver

The drm/amdgpu driver supports all AMD Radeon GPUs based on the Graphics Core Next (GCN), Radeon DNA (RDNA), and Compute DNA (CDNA) architectures.

10.1.1 Module Parameters

The amdgpu driver supports the following module parameters:

vramlimit (int)

Restrict the total amount of VRAM in MiB for testing. The default is 0 (Use full VRAM).

vis_vramlimit (int)

Restrict the amount of CPU visible VRAM in MiB for testing. The default is 0 (Use full CPU visible VRAM).

gartsize (uint)

Restrict the size of GART (for kernel use) in Mib (32, 64, etc.) for testing. The default is -1 (The size depends on asic).

gttsize (int)

Restrict the size of GTT domain (for userspace use) in MiB for testing. The default is -1 (Use 1/2 RAM, minimum value is 3GB).

moverate (int)

Set maximum buffer migration rate in MB/s. The default is -1 (8 MB/s).

audio (int)

Set HDMI/DPAudio. Only affects non-DC display handling. The default is -1 (Enabled), set 0 to disabled it.

disp_priority (int)

Set display Priority (1 = normal, 2 = high). Only affects non-DC display handling. The default is 0 (auto).

hw_i2c (int)

To enable hw i2c engine. Only affects non-DC display handling. The default is 0 (Disabled).

pcie_gen2 (int)

To disable PCIE Gen2/3 mode (0 = disable, 1 = enable). The default is -1 (auto, enabled).

msi (int)

To disable Message Signaled Interrupts (MSI) functionality (1 = enable, 0 = disable). The default is -1 (auto, enabled).

lockup_timeout (string)

Set GPU scheduler timeout value in ms.

The format can be [Non-Compute] or [GFX,Compute,SDMA,Video]. That is there can be one or multiple values specified. 0 and negative values are invalidated. They will be adjusted to the default timeout.

- With one value specified, the setting will apply to all non-compute jobs.
- With multiple values specified, the first one will be for GFX. The second one is for Compute. The third and fourth ones are for SDMA and Video.

By default(with no lockup_timeout settings), the timeout for all non-compute(GFX, SDMA and Video) jobs is 10000. The timeout for compute is 60000.

dpm (int)

Override for dynamic power management setting (0 = disable, 1 = enable) The default is -1 (auto).

fw_load_type (int)

Set different firmware loading type for debugging, if supported. Set to 0 to force direct loading if supported by the ASIC. Set to -1 to select the default loading mode for the ASIC, as defined by the driver. The default is -1 (auto).

aspm (int)

To disable ASPM (1 = enable, 0 = disable). The default is -1 (auto, enabled).

runpm (int)

Override for runtime power management control for dGPUs. The amdgpu driver can dynamically power down the dGPUs when they are idle if supported. The default is -1 (auto enable). Setting the value to 0 disables this functionality. Setting the value to -2 is auto enabled with power down when displays are attached.

ip_block_mask (uint)

Override what IP blocks are enabled on the GPU. Each GPU is a collection of IP blocks (gfx, display, video, etc.). Use this parameter to disable specific blocks. Note that the IP blocks do not have a fixed index. Some asics may not have some IPs or may include multiple instances of an IP so the ordering varies from asic to asic. See the driver output in the kernel log for the list of IPs on the asic. The default is 0xffffffff (enable all blocks on a device).

bapm (int)

Bidirectional Application Power Management (BAPM) used to dynamically share TDP between CPU and GPU. Set value 0 to disable it. The default -1 (auto, enabled)

deep_color (int)

Set 1 to enable Deep Color support. Only affects non-DC display handling. The default is 0 (disabled).

vm_size (int)

Override the size of the GPU's per client virtual address space in GiB. The default is -1 (automatic for each asic).

vm_fragment_size (int)

Override VM fragment size in bits (4, 5, etc. 4 = 64K, 9 = 2M). The default is -1 (automatic for each asic).

vm_block_size (int)

Override VM page table size in bits (default depending on vm_size and hw setup). The default is -1 (automatic for each asic).

vm_fault_stop (int)

Stop on VM fault for debugging (0 = never, 1 = print first, 2 = always). The default is 0 (No stop).

vm_debug (int)

Debug VM handling (0 = disabled, 1 = enabled). The default is 0 (Disabled).

vm_update_mode (int)

Override VM update mode. VM updated by using CPU (0 = never, 1 = Graphics only, 2 = Compute only, 3 = Both). The default is -1 (Only in large BAR(LB) systems Compute VM tables will be updated by CPU, otherwise 0, never).

exp_hw_support (int)

Enable experimental hw support (1 = enable). The default is 0 (disabled).

dc (int)

Disable/Enable Display Core driver for debugging (1 = enable, 0 = disable). The default is -1 (automatic for each asic).

sched_jobs (int)

Override the max number of jobs supported in the sw queue. The default is 32.

sched_hw_submission (int)

Override the max number of HW submissions. The default is 2.

ppfeaturemask (hexint)

Override power features enabled. See enum PP_FEATURE_MASK in drivers/gpu/drm/amd/include/amd_shared.h. The default is the current set of stable power features.

forcelongtraining (uint)

Force long memory training in resume. The default is zero, indicates short training in resume.

pcie_gen_cap (uint)

Override PCIE gen speed capabilities. See the CAIL flags in drivers/gpu/drm/amd/include/amd_pcie.h. The default is 0 (automatic for each asic).

pcie_lane_cap (uint)

Override PCIE lanes capabilities. See the CAIL flags in drivers/gpu/drm/amd/include/amd_pcie.h. The default is 0 (automatic for each asic).

cg_mask (ulong)

Override Clockgating features enabled on GPU (0 = disable clock gating). See the AMD_CG_SUPPORT flags in drivers/gpu/drm/amd/include/amd_shared.h. The default is 0xffffffff (all enabled).

pg_mask (uint)

Override Powergating features enabled on GPU (0 = disable power gating). See the AMD_PG_SUPPORT flags in drivers/gpu/drm/amd/include/amd_shared.h. The default is 0xffffffff (all enabled).

sdma_phase_quantum (uint)

Override SDMA context switch phase quantum (x 1K GPU clock cycles, 0 = no change). The default is 32.

disable_cu (charp)

Set to disable CUs (It's set like se.sh.cu,...). The default is NULL.

virtual_display (charp)

Set to enable virtual display feature. This feature provides a virtual display hardware on headless boards or in virtualized environments. It will be set like xxxx:xx:xx.x,x;xxxx:xx:xx.x,x. It's the pci address of the device, plus the number of crtcs to expose. E.g., 0000:26:00.0,4 would enable 4 virtual crtcs on the pci device at 26:00.0. The default is NULL.

lbpw (int)

Override Load Balancing Per Watt (LBPW) support (1 = enable, 0 = disable). The default is -1 (auto, enabled).

gpu_recovery (int)

Set to enable GPU recovery mechanism (1 = enable, 0 = disable). The default is -1 (auto, disabled except SRIOV).

emu_mode (int)

Set value 1 to enable emulation mode. This is only needed when running on an emulator. The default is 0 (disabled).

ras_enable (int)

Enable RAS features on the GPU (0 = disable, 1 = enable, -1 = auto (default))

ras_mask (uint)

Mask of RAS features to enable (default 0xffffffff), only valid when ras_enable == 1 See the flags in drivers/gpu/drm/amd/amdgpu/amdgpu_ras.h

timeout_fatal_disable (bool)

Disable Watchdog timeout fatal error event

timeout_period (uint)

Modify the watchdog timeout max_cycles as (1 << period)

si_support (int)

Set SI support driver. This parameter works after set config CONFIG_DRM_AMDGPU_SI. For SI asic, when radeon driver is enabled, set value 0 to use radeon driver, while set value 1 to use amdgpu driver. The default is using radeon driver when it available, otherwise using amdgpu driver.

cik_support (int)

Set CIK support driver. This parameter works after set config CONFIG_DRM_AMDGPU_CIK. For CIK asic, when radeon driver is enabled, set value 0 to use radeon driver, while set value 1 to use amdgpu driver. The default is using radeon driver when it available, otherwise using amdgpu driver.

smu_memory_pool_size (uint)

It is used to reserve gtt for smu debug usage, setting value 0 to disable it. The actual size is value * 256MiB. E.g. 0x1 = 256Mbyte, 0x2 = 512Mbyte, 0x4 = 1 Gbyte, 0x8 = 2GByte. The default is 0 (disabled).

async_gfx_ring (int)

It is used to enable gfx rings that could be configured with different priorities or equal priorities

mcbp (int)

It is used to enable mid command buffer preemption. (0 = disabled, 1 = enabled, -1 auto (default))

discovery (int)

Allow driver to discover hardware IP information from IP Discovery table at the top of VRAM. (-1 = auto (default), 0 = disabled, 1 = enabled, 2 = use ip_discovery table from file)

mes (int)

Enable Micro Engine Scheduler. This is a new hw scheduling engine for gfx, sdma, and compute. (0 = disabled (default), 1 = enabled)

mes_kiq (int)

Enable Micro Engine Scheduler KIQ. This is a new engine pipe for kiq. (0 = disabled (default), 1 = enabled)

noretry (int)

Disable XNACK retry in the SQ by default on GFXv9 hardware. On ASICs that do not support per-process XNACK this also disables retry page faults. (0 = retry enabled, 1 = retry disabled, -1 auto (default))

force_asic_type (int)

A non negative value used to specify the asic type for all supported GPUs.

use_xgmi_p2p (int)

Enables/disables XGMI P2P interface (0 = disable, 1 = enable).

sched_policy (int)

Set scheduling policy. Default is HWS(hardware scheduling) with over-subscription. Setting 1 disables over-subscription. Setting 2 disables HWS and statically assigns queues to HQDs.

hws_max_conc_proc (int)

Maximum number of processes that HWS can schedule concurrently. The maximum is the number of VMIDs assigned to the HWS, which is also the default.

cwsr_enable (int)

CWSR(compute wave store and resume) allows the GPU to preempt shader execution in the middle of a compute wave. Default is 1 to enable this feature. Setting 0 disables it.

max_num_of_queues_per_device (int)

Maximum number of queues per device. Valid setting is between 1 and 4096. Default is 4096.

send_sigterm (int)

Send sigterm to HSA process on unhandled exceptions. Default is not to send sigterm but just print errors on dmesg. Setting 1 enables sending sigterm.

debug_largebar (int)

Set debug_largebar as 1 to enable simulating large-bar capability on non-large bar system. This limits the VRAM size reported to ROCm applications to the visible size, usually 256MB. Default value is 0, disabled.

halt_if_hws_hang (int)

Halt if HWS hang is detected. Default value, 0, disables the halt on hang. Setting 1 enables halt on hang.

hws_gws_support(bool)

Assume that HWS supports GWS barriers regardless of what firmware version check says. Default value: false (rely on MEC2 firmware version check).

queue_preemption_timeout_ms (int)

queue preemption timeout in ms (1 = Minimum, 9000 = default)

debug_evictions(bool)

Enable extra debug messages to help determine the cause of evictions

no_system_mem_limit(bool)

Disable system memory limit, to support multiple process shared memory

no_queue_eviction_on_vm_fault (int)

If set, process queues will not be evicted on gpuvn fault. This is to keep the wavefront context for debugging (0 = queue eviction, 1 = no queue eviction). The default is 0 (queue eviction).

mtype_local (int)**pcie_p2p (bool)**

Enable PCIe P2P (requires large-BAR). Default value: true (on)

dcfeaturemask (uint)

Override display features enabled. See enum DC_FEATURE_MASK in drivers/gpu/drm/amd/include/amd_shared.h. The default is the current set of stable display features.

dcdebugmask (uint)

Override display features enabled. See enum `DC_DEBUG_MASK` in `drivers/gpu/drm/amd/include/amd_shared.h`.

abmlevel (uint)

Override the default ABM (Adaptive Backlight Management) level used for DC enabled hardware. Requires DMCU to be supported and loaded. Valid levels are 0-4. A value of 0 indicates that ABM should be disabled by default. Values 1-4 control the maximum allowable brightness reduction via the ABM algorithm, with 1 being the least reduction and 4 being the most reduction.

Defaults to 0, or disabled. Userspace can still override this level later after boot.

tmz (int)

Trusted Memory Zone (TMZ) is a method to protect data being written to or read from memory. The default value: 0 (off). TODO: change to auto till it is completed.

reset_method (int)

GPU reset method (-1 = auto (default), 0 = legacy, 1 = mode0, 2 = mode1, 3 = mode2, 4 = baco)

bad_page_threshold (int) Bad page threshold is specifies the

threshold value of faulty pages detected by RAS ECC, which may result in the GPU entering bad status when the number of total faulty pages by ECC exceeds the threshold value.

vcnfw_log (int)

Enable vcnfw log output for debugging, the default is disabled.

sg_display (int)

Disable S/G (scatter/gather) display (i.e., display from system memory). This option is only relevant on APUs. Set this option to 0 to disable S/G display if you experience flickering or other issues under memory pressure and report the issue.

smu_pptable_id (int)

Used to override pptable id. id = 0 use VBIOS pptable. id > 0 use the soft pptable with specified id.

partition_mode (int)

Used to override the default SPX mode.

enforce_isolation (bool)

enforce process isolation between graphics and compute via using the same reserved vmid.

```
void amdgpu_drv_delayed_reset_work_handler(struct work_struct *work)
    work handler for reset
```

Parameters

```
struct work_struct *work
    work_struct.
```

10.1.2 Core Driver Infrastructure

GPU Hardware Structure

Each ASIC is a collection of hardware blocks. We refer to them as “IPs” (Intellectual Property blocks). Each IP encapsulates certain functionality. IPs are versioned and can also be mixed and matched. E.g., you might have two different ASICs that both have System DMA (SDMA) 5.x IPs. The driver is arranged by IPs. There are driver components to handle the initialization and operation of each IP. There are also a bunch of smaller IPs that don’t really need much if any driver interaction. Those end up getting lumped into the common stuff in the soc files. The soc files (e.g., vi.c, soc15.c nv.c) contain code for aspects of the SoC itself rather than specific IPs. E.g., things like GPU resets and register access functions are SoC dependent.

An APU contains more than just CPU and GPU, it also contains all of the platform stuff (audio, usb, gpio, etc.). Also, a lot of components are shared between the CPU, platform, and the GPU (e.g., SMU, PSP, etc.). Specific components (CPU, GPU, etc.) usually have their interface to interact with those common components. For things like S0i3 there is a ton of coordination required across all the components, but that is probably a bit beyond the scope of this section.

With respect to the GPU, we have the following major IPs:

GMC (Graphics Memory Controller)

This was a dedicated IP on older pre-vega chips, but has since become somewhat decentralized on vega and newer chips. They now have dedicated memory hubs for specific IPs or groups of IPs. We still treat it as a single component in the driver however since the programming model is still pretty similar. This is how the different IPs on the GPU get the memory (VRAM or system memory). It also provides the support for per process GPU virtual address spaces.

IH (Interrupt Handler)

This is the interrupt controller on the GPU. All of the IPs feed their interrupts into this IP and it aggregates them into a set of ring buffers that the driver can parse to handle interrupts from different IPs.

PSP (Platform Security Processor)

This handles security policy for the SoC and executes trusted applications, and validates and loads firmwares for other blocks.

SMU (System Management Unit)

This is the power management microcontroller. It manages the entire SoC. The driver interacts with it to control power management features like clocks, voltages, power rails, etc.

DCN (Display Controller Next)

This is the display controller. It handles the display hardware. It is described in more details in [Display Core](#).

SDMA (System DMA)

This is a multi-purpose DMA engine. The kernel driver uses it for various things including paging and GPU page table updates. It’s also exposed to userspace for use by user mode drivers (OpenGL, Vulkan, etc.)

GC (Graphics and Compute)

This is the graphics and compute engine, i.e., the block that encompasses the 3D pipeline and and shader blocks. This is by far the largest block on the GPU. The 3D pipeline has tons of sub-blocks. In addition to that, it also contains the CP microcontrollers (ME, PFP,

CE, MEC) and the RLC microcontroller. It's exposed to userspace for user mode drivers (OpenGL, Vulkan, OpenCL, etc.)

VCN (Video Core Next)

This is the multi-media engine. It handles video and image encode and decode. It's exposed to userspace for user mode drivers (VA-API, OpenMAX, etc.)

Graphics and Compute Microcontrollers**CP (Command Processor)**

The name for the hardware block that encompasses the front end of the GFX/Compute pipeline. Consists mainly of a bunch of microcontrollers (PFP, ME, CE, MEC). The firmware that runs on these microcontrollers provides the driver interface to interact with the GFX/Compute engine.

MEC (MicroEngine Compute)

This is the microcontroller that controls the compute queues on the GFX/compute engine.

MES (MicroEngine Scheduler)

This is a new engine for managing queues. This is currently unused.

RLC (RunList Controller)

This is another microcontroller in the GFX/Compute engine. It handles power management related functionality within the GFX/Compute engine. The name is a vestige of old hardware where it was originally added and doesn't really have much relation to what the engine does now.

Driver Structure

In general, the driver has a list of all of the IPs on a particular SoC and for things like init/fini/suspend/resume, more or less just walks the list and handles each IP.

Some useful constructs:

KIQ (Kernel Interface Queue)

This is a control queue used by the kernel driver to manage other gfx and compute queues on the GFX/compute engine. You can use it to map/unmap additional queues, etc.

IB (Indirect Buffer)

A command buffer for a particular engine. Rather than writing commands directly to the queue, you can write the commands into a piece of memory and then put a pointer to the memory into the queue. The hardware will then follow the pointer and execute the commands in the memory, then returning to the rest of the commands in the ring.

Memory Domains

AMDGPU_GEM_DOMAIN_CPU System memory that is not GPU accessible. Memory in this pool could be swapped out to disk if there is pressure.

AMDGPU_GEM_DOMAIN_GTT GPU accessible system memory, mapped into the GPU's virtual address space via gart. Gart memory linearizes non-contiguous pages of system memory, allows GPU access system memory in a linearized fashion.

AMDGPU_GEM_DOMAIN_VRAM Local video memory. For APUs, it is memory carved out by the BIOS.

AMDGPU_GEM_DOMAIN_GDS Global on-chip data storage used to share data across shader threads.

AMDGPU_GEM_DOMAIN_GWS Global wave sync, used to synchronize the execution of all the waves on a device.

AMDGPU_GEM_DOMAIN_OA Ordered append, used by 3D or Compute engines for appending data.

AMDGPU_GEM_DOMAIN_DOORBELL Doorbell. It is an MMIO region for signalling user mode queues.

Buffer Objects

This defines the interfaces to operate on an `amdgpu_bo` buffer object which represents memory used by driver (VRAM, system memory, etc.). The driver provides DRM/GEM APIs to userspace. DRM/GEM APIs then use these interfaces to create/destroy/set buffer object which are then managed by the kernel TTM memory manager. The interfaces are also used internally by kernel clients, including gfx, uvd, etc. for kernel managed allocations used by the GPU.

```
bool amdgpu_bo_is_amdgpu_bo(struct ttm_buffer_object *bo)
    check if the buffer object is an amdgpu_bo
```

Parameters

```
struct ttm_buffer_object *bo
    buffer object to be checked
```

Description

Uses destroy function associated with the object to determine if this is an `amdgpu_bo`.

Return

true if the object belongs to `amdgpu_bo`, false if not.

```
void amdgpu_bo_placement_from_domain(struct amdgpu_bo *abo, u32 domain)
    set buffer's placement
```

Parameters

```
struct amdgpu_bo *abo
    amdgpu_bo buffer object whose placement is to be set

u32 domain
    requested domain
```

Description

Sets buffer's placement according to requested domain and the buffer's flags.

```
int amdgpu_bo_create_reserved(struct amdgpu_device *adev, unsigned long size, int align,
                             u32 domain, struct amdgpu_bo **bo_ptr, u64 *gpu_addr,
                             void **cpu_addr)
```

create reserved BO for kernel use

Parameters

struct amdgpu_device *adev
amdgpu device object

unsigned long size
size for the new BO

int align
alignment for the new BO

u32 domain
where to place it

struct amdgpu_bo **bo_ptr
used to initialize BOs in structures

u64 *gpu_addr
GPU addr of the pinned BO

void **cpu_addr
optional CPU address mapping

Description

Allocates and pins a BO for kernel internal use, and returns it still reserved.

Note

For bo_ptr new BO is only created if bo_ptr points to NULL.

Return

0 on success, negative error code otherwise.

```
int amdgpu_bo_create_kernel(struct amdgpu_device *adev, unsigned long size, int align, u32
                           domain, struct amdgpu_bo **bo_ptr, u64 *gpu_addr, void
                           **cpu_addr)
```

create BO for kernel use

Parameters

struct amdgpu_device *adev
amdgpu device object

unsigned long size
size for the new BO

int align
alignment for the new BO

u32 domain
where to place it

struct amdgpu_bo **bo_ptr
used to initialize BOs in structures

u64 *gpu_addr

GPU addr of the pinned BO

void **cpu_addr

optional CPU address mapping

Description

Allocates and pins a BO for kernel internal use.

Note

For `bo_ptr` new BO is only created if `bo_ptr` points to NULL.

Return

0 on success, negative error code otherwise.

int amdgpu_bo_create_kernel_at(struct amdgpu_device *adev, uint64_t offset, uint64_t size,
struct amdgpu_bo **bo_ptr, void **cpu_addr)

create BO for kernel use at specific location

Parameters

struct amdgpu_device *adev

amdgpu device object

uint64_t offset

offset of the BO

uint64_t size

size of the BO

struct amdgpu_bo **bo_ptr

used to initialize BOs in structures

void **cpu_addr

optional CPU address mapping

Description

Creates a kernel BO at a specific offset in VRAM.

Return

0 on success, negative error code otherwise.

void amdgpu_bo_free_kernel(struct amdgpu_bo **bo, u64 *gpu_addr, void **cpu_addr)

free BO for kernel use

Parameters

struct amdgpu_bo **bo

amdgpu BO to free

u64 *gpu_addr

pointer to where the BO's GPU memory space address was stored

void **cpu_addr

pointer to where the BO's CPU memory space address was stored

Description

unmaps and unpin a BO for kernel internal use.

```
int amdgpu_bo_create(struct amdgpu_device *adev, struct amdgpu_bo_param *bp, struct
                    amdgpu_bo **bo_ptr)
```

create an amdgpu_bo buffer object

Parameters

```
struct amdgpu_device *adev
```

amdgpu device object

```
struct amdgpu_bo_param *bp
```

parameters to be used for the buffer object

```
struct amdgpu_bo **bo_ptr
```

pointer to the buffer object pointer

Description

Creates an amdgpu_bo buffer object.

Return

0 for success or a negative error code on failure.

```
int amdgpu_bo_create_user(struct amdgpu_device *adev, struct amdgpu_bo_param *bp,
                         struct amdgpu_bo_user **ubo_ptr)
```

create an amdgpu_bo_user buffer object

Parameters

```
struct amdgpu_device *adev
```

amdgpu device object

```
struct amdgpu_bo_param *bp
```

parameters to be used for the buffer object

```
struct amdgpu_bo_user **ubo_ptr
```

pointer to the buffer object pointer

Description

Create a BO to be used by user application;

Return

0 for success or a negative error code on failure.

```
int amdgpu_bo_create_vm(struct amdgpu_device *adev, struct amdgpu_bo_param *bp, struct
                      amdgpu_bo_vm **vmbo_ptr)
```

create an amdgpu_bo_vm buffer object

Parameters

```
struct amdgpu_device *adev
```

amdgpu device object

```
struct amdgpu_bo_param *bp
```

parameters to be used for the buffer object

struct amdgpu_bo_vm **vmbo_ptr
pointer to the buffer object pointer

Description

Create a BO to be for GPUVM.

Return

0 for success or a negative error code on failure.

void **amdgpu_bo_add_to_shadow_list**(struct amdgpu_bo_vm *vmbo)
add a BO to the shadow list

Parameters

struct amdgpu_bo_vm *vmbo
BO that will be inserted into the shadow list

Description

Insert a BO to the shadow list.

int **amdgpu_bo_restore_shadow**(struct amdgpu_bo *shadow, struct dma_fence **fence)
restore an amdgpu_bo shadow

Parameters

struct amdgpu_bo *shadow
amdgpu_bo shadow to be restored

struct dma_fence **fence
dma_fence associated with the operation

Description

Copies a buffer object's shadow content back to the object. This is used for recovering a buffer from its shadow in case of a gpu reset where vram context may be lost.

Return

0 for success or a negative error code on failure.

int **amdgpu_bo_kmap**(struct amdgpu_bo *bo, void **ptr)
map an amdgpu_bo buffer object

Parameters

struct amdgpu_bo *bo
amdgpu_bo buffer object to be mapped

void **ptr
kernel virtual address to be returned

Description

Calls ttm_bo_kmap() to set up the kernel virtual mapping; calls [amdgpu_bo_kptr\(\)](#) to get the kernel virtual address.

Return

0 for success or a negative error code on failure.

void ***amdgpu_bo_kptr**(struct amdgpu_bo *bo)
returns a kernel virtual address of the buffer object

Parameters

struct amdgpu_bo *bo
amdgpu_bo buffer object

Description

Calls `ttm_kmap_obj_virtual()` to get the kernel virtual address

Return

the virtual address of a buffer object area.

void **amdgpu_bo_kunmap**(struct amdgpu_bo *bo)
unmap an amdgpu_bo buffer object

Parameters

struct amdgpu_bo *bo
amdgpu_bo buffer object to be unmapped

Description

Unmaps a kernel map set up by [*amdgpu_bo_kmap\(\)*](#).

struct amdgpu_bo ***amdgpu_bo_ref**(struct amdgpu_bo *bo)
reference an amdgpu_bo buffer object

Parameters

struct amdgpu_bo *bo
amdgpu_bo buffer object

Description

References the contained `ttm_buffer_object`.

Return

a refcounted pointer to the amdgpu_bo buffer object.

void **amdgpu_bo_unref**(struct amdgpu_bo **bo)
unreference an amdgpu_bo buffer object

Parameters

struct amdgpu_bo **bo
amdgpu_bo buffer object

Description

Unreferences the contained `ttm_buffer_object` and clear the pointer

int **amdgpu_bo_pin_restricted**(struct amdgpu_bo *bo, u32 domain, u64 min_offset, u64 max_offset)
pin an amdgpu_bo buffer object

Parameters

struct amdgpu_bo *bo
amdgpu_bo buffer object to be pinned

u32 domain
domain to be pinned to

u64 min_offset
the start of requested address range

u64 max_offset
the end of requested address range

Description

Pins the buffer object according to requested domain and address range. If the memory is unbound gart memory, binds the pages into gart table. Adjusts pin_count and pin_size accordingly.

Pinning means to lock pages in memory along with keeping them at a fixed offset. It is required when a buffer can not be moved, for example, when a display buffer is being scanned out.

Compared with [amdgpu_bo_pin\(\)](#), this function gives more flexibility on where to pin a buffer if there are specific restrictions on where a buffer must be located.

Return

0 for success or a negative error code on failure.

int **amdgpu_bo_pin**(struct amdgpu_bo *bo, u32 domain)
pin an amdgpu_bo buffer object

Parameters

struct amdgpu_bo *bo
amdgpu_bo buffer object to be pinned

u32 domain
domain to be pinned to

Description

A simple wrapper to [amdgpu_bo_pin_restricted\(\)](#). Provides a simpler API for buffers that do not have any strict restrictions on where a buffer must be located.

Return

0 for success or a negative error code on failure.

void **amdgpu_bo_unpin**(struct amdgpu_bo *bo)
unpin an amdgpu_bo buffer object

Parameters

struct amdgpu_bo *bo
amdgpu_bo buffer object to be unpinned

Description

Decreases the pin_count, and clears the flags if pin_count reaches 0. Changes placement and pin size accordingly.

Return

0 for success or a negative error code on failure.

int **amdgpu_bo_init**(struct amdgpu_device *adev)
 initialize memory manager

Parameters

struct amdgpu_device *adev
 amdgpu device object

Description

Calls `amdgpu_ttm_init()` to initialize amdgpu memory manager.

Return

0 for success or a negative error code on failure.

void **amdgpu_bo_fini**(struct amdgpu_device *adev)
 tear down memory manager

Parameters

struct amdgpu_device *adev
 amdgpu device object

Description

Reverses `amdgpu_bo_init()` to tear down memory manager.

int **amdgpu_bo_set_tiling_flags**(struct amdgpu_bo *bo, u64 tiling_flags)
 set tiling flags

Parameters

struct amdgpu_bo *bo
 amdgpu_bo buffer object

u64 tiling_flags
 new flags

Description

Sets buffer object's tiling flags with the new one. Used by GEM ioctl or kernel driver to set the tiling flags on a buffer.

Return

0 for success or a negative error code on failure.

void **amdgpu_bo_get_tiling_flags**(struct amdgpu_bo *bo, u64 *tiling_flags)
 get tiling flags

Parameters

struct amdgpu_bo *bo
 amdgpu_bo buffer object

u64 *tiling_flags
 returned flags

Description

Gets buffer object's tiling flags. Used by GEM ioctl or kernel driver to set the tiling flags on a buffer.

```
int amdgpu_bo_set_metadata(struct amdgpu_bo *bo, void *metadata, u32 metadata_size,
                           uint64_t flags)
```

set metadata

Parameters

struct amdgpu_bo *bo
amdgpu_bo buffer object

void *metadata
new metadata

u32 metadata_size
size of the new metadata

uint64_t flags
flags of the new metadata

Description

Sets buffer object's metadata, its size and flags. Used via GEM ioctl.

Return

0 for success or a negative error code on failure.

```
int amdgpu_bo_get_metadata(struct amdgpu_bo *bo, void *buffer, size_t buffer_size, uint32_t
                           *metadata_size, uint64_t *flags)
```

get metadata

Parameters

struct amdgpu_bo *bo
amdgpu_bo buffer object

void *buffer
returned metadata

size_t buffer_size
size of the buffer

uint32_t *metadata_size
size of the returned metadata

uint64_t *flags
flags of the returned metadata

Description

Gets buffer object's metadata, its size and flags. `buffer_size` shall not be less than `metadata_size`. Used via GEM ioctl.

Return

0 for success or a negative error code on failure.

void **amdgpu_bo_move_notify**(struct ttm_buffer_object *bo, bool evict, struct *ttm_resource* *new_mem)

notification about a memory move

Parameters

struct ttm_buffer_object *bo

pointer to a buffer object

bool evict

if this move is evicting the buffer from the graphics address space

struct ttm_resource *new_mem

new resource for backing the BO

Description

Marks the corresponding `amdgpu_bo` buffer object as invalid, also performs bookkeeping. TTM driver callback which is called when ttm moves a buffer.

void **amdgpu_bo_release_notify**(struct ttm_buffer_object *bo)

notification about a BO being released

Parameters

struct ttm_buffer_object *bo

pointer to a buffer object

Description

Wipes VRAM buffers whose contents should not be leaked before the memory is released.

vm_fault_t **amdgpu_bo_fault_reserve_notify**(struct ttm_buffer_object *bo)

notification about a memory fault

Parameters

struct ttm_buffer_object *bo

pointer to a buffer object

Description

Notifies the driver we are taking a fault on this BO and have reserved it, also performs bookkeeping. TTM driver callback for dealing with vm faults.

Return

0 for success or a negative error code on failure.

void **amdgpu_bo_fence**(struct `amdgpu_bo` *bo, struct `dma_fence` *fence, bool shared)

add fence to buffer object

Parameters

struct `amdgpu_bo` *bo

buffer object in question

struct `dma_fence` *fence

fence to add

bool shared

true if fence should be added shared

int **amdgpu_bo_sync_wait_resv**(struct amdgpu_device *adev, struct dma_resv *resv, enum amdgpu_sync_mode sync_mode, void *owner, bool intr)

Wait for BO reservation fences

Parameters

struct amdgpu_device *adev
amdgpu device pointer

struct dma_resv *resv
reservation object to sync to

enum amdgpu_sync_mode sync_mode
synchronization mode

void *owner
fence owner

bool intr
Whether the wait is interruptible

Description

Extract the fences from the reservation object and waits for them to finish.

Return

0 on success, errno otherwise.

int **amdgpu_bo_sync_wait**(struct amdgpu_bo *bo, void *owner, bool intr)
Wrapper for amdgpu_bo_sync_wait_resv

Parameters

struct amdgpu_bo *bo
buffer object to wait for

void *owner
fence owner

bool intr
Whether the wait is interruptible

Description

Wrapper to wait for fences in a BO.

Return

0 on success, errno otherwise.

u64 **amdgpu_bo_gpu_offset**(struct amdgpu_bo *bo)
return GPU offset of bo

Parameters

struct amdgpu_bo *bo
amdgpu object for which we query the offset

Note

object should either be pinned or reserved when calling this function, it might be useful to add check for this for debugging.

Return

current GPU offset of the object.

```
u64 amdgpu_bo_gpu_offset_no_check(struct amdgpu_bo *bo)
    return GPU offset of bo
```

Parameters

```
struct amdgpu_bo *bo
    amdgpu object for which we query the offset
```

Return

current GPU offset of the object without raising warnings.

```
uint32_t amdgpu_bo_get_preferred_domain(struct amdgpu_device *adev, uint32_t domain)
    get preferred domain
```

Parameters

```
struct amdgpu_device *adev
    amdgpu device object
```

```
uint32_t domain
    allowed memory domains
```

Return

Which of the allowed domains is preferred for allocating the BO.

```
u64 amdgpu_bo_print_info(int id, struct amdgpu_bo *bo, struct seq_file *m)
    print BO info in debugfs file
```

Parameters

```
int id
    Index or Id of the BO
```

```
struct amdgpu_bo *bo
    Requested BO for printing info
```

```
struct seq_file *m
    debugfs file
```

Description

Print BO information in debugfs file

Return

Size of the BO in bytes.

PRIME Buffer Sharing

The following callback implementations are used for *sharing GEM buffer objects between different devices via PRIME*.

int **amdgpu_dma_buf_attach**(struct dma_buf *dmabuf, struct dma_buf_attachment *attach)
dma_buf_ops.attach implementation

Parameters

struct dma_buf *dmabuf
DMA-buf where we attach to

struct dma_buf_attachment *attach
attachment to add

Description

Add the attachment as user to the exported DMA-buf.

void **amdgpu_dma_buf_detach**(struct dma_buf *dmabuf, struct dma_buf_attachment *attach)
dma_buf_ops.detach implementation

Parameters

struct dma_buf *dmabuf
DMA-buf where we remove the attachment from

struct dma_buf_attachment *attach
the attachment to remove

Description

Called when an attachment is removed from the DMA-buf.

int **amdgpu_dma_buf_pin**(struct dma_buf_attachment *attach)
dma_buf_ops.pin implementation

Parameters

struct dma_buf_attachment *attach
attachment to pin down

Description

Pin the BO which is backing the DMA-buf so that it can't move any more.

void **amdgpu_dma_buf_unpin**(struct dma_buf_attachment *attach)
dma_buf_ops.unpin implementation

Parameters

struct dma_buf_attachment *attach
attachment to unpin

Description

Unpin a previously pinned BO to make it movable again.

struct sg_table ***amdgpu_dma_buf_map**(struct dma_buf_attachment *attach, enum
dma_data_direction dir)
dma_buf_ops.map_dma_buf implementation

Parameters

struct dma_buf_attachment *attach
DMA-buf attachment

enum dma_data_direction dir
DMA direction

Description

Makes sure that the shared DMA buffer can be accessed by the target device. For now, simply pins it to the GTT domain, where it should be accessible by all DMA devices.

Return

sg_table filled with the DMA addresses to use or ERR_PRT with negative error code.

void **amdgpu_dma_buf_unmap**(struct dma_buf_attachment *attach, struct sg_table *sgt, enum dma_data_direction dir)
dma_buf_ops.unmap_dma_buf implementation

Parameters

struct dma_buf_attachment *attach
DMA-buf attachment

struct sg_table *sgt
sg_table to unmap

enum dma_data_direction dir
DMA direction

Description

This is called when a shared DMA buffer no longer needs to be accessible by another device. For now, simply unpins the buffer from GTT.

int **amdgpu_dma_buf_begin_cpu_access**(struct *dma_buf* *dma_buf, enum dma_data_direction direction)
dma_buf_ops.begin_cpu_access implementation

Parameters

struct dma_buf *dma_buf
Shared DMA buffer

enum dma_data_direction direction
Direction of DMA transfer

Description

This is called before CPU access to the shared DMA buffer's memory. If it's a read access, the buffer is moved to the GTT domain if possible, for optimal CPU read performance.

Return

0 on success or a negative error code on failure.

struct dma_buf ***amdgpu_gem_prime_export**(struct *drm_gem_object* *gobj, int flags)
drm_driver.gem_prime_export implementation

Parameters

struct *drm_gem_object* *gobj
GEM BO

int flags
Flags such as `DRM_CLOEXEC` and `DRM_RDWR`.

Description

The main work is done by the *drm_gem_prime_export* helper.

Return

Shared DMA buffer representing the GEM BO from the given device.

struct *drm_gem_object* *amdgpu_dma_buf_create_obj(struct *drm_device* *dev, struct *dma_buf* *dma_buf)
create BO for DMA-buf import

Parameters

struct *drm_device* *dev
DRM device

struct *dma_buf* *dma_buf
DMA-buf

Description

Creates an empty SG BO for DMA-buf import.

Return

A new GEM BO of the given DRM device, representing the memory described by the given DMA-buf attachment and scatter/gather table.

void **amdgpu_dma_buf_move_notify**(struct *dma_buf_attachment* *attach)
attach.move_notify implementation

Parameters

struct *dma_buf_attachment* *attach
the DMA-buf attachment

Description

Invalidate the DMA-buf attachment, making sure that the we re-create the mapping before the next use.

struct *drm_gem_object* *amdgpu_gem_prime_import(struct *drm_device* *dev, struct *dma_buf* *dma_buf)
drm_driver.gem_prime_import implementation

Parameters

struct *drm_device* *dev
DRM device

struct *dma_buf* *dma_buf
Shared DMA buffer

Description

Import a dma_buf into a the driver and potentially create a new GEM object.

Return

GEM BO representing the shared DMA buffer for the given device.

bool **amdgpu_dmabuf_is_xgmi_accessible**(struct amdgpu_device *adev, struct amdgpu_bo *bo)

Check if xgmi available for P2P transfer

Parameters

struct amdgpu_device *adev
amdgpu_device pointer of the importer

struct amdgpu_bo *bo
amdgpu buffer object

Return

True if dmabuf accessible over xgmi, false otherwise.

MMU Notifier

For coherent userptr handling registers an MMU notifier to inform the driver about updates on the page tables of a process.

When somebody tries to invalidate the page tables we block the update until all operations on the pages in question are completed, then those pages are marked as accessed and also dirty if it wasn't a read only access.

New command submissions using the userptrs in question are delayed until all page table invalidation are completed and we once more see a coherent process address space.

bool **amdgpu_hmm_invalidate_gfx**(struct mmu_interval_notifier *mni, const struct mmu_notifier_range *range, unsigned long cur_seq)
callback to notify about mm change

Parameters

struct mmu_interval_notifier *mni
the range (mm) is about to update

const struct mmu_notifier_range *range
details on the invalidation

unsigned long cur_seq
Value to pass to mmu_interval_set_seq()

Description

Block for operations on BOs to finish and mark pages as accessed and potentially dirty.

bool **amdgpu_hmm_invalidate_hsa**(struct mmu_interval_notifier *mni, const struct mmu_notifier_range *range, unsigned long cur_seq)
callback to notify about mm change

Parameters

struct mmu_interval_notifier *mni

the range (mm) is about to update

const struct mmu_notifier_range *range

details on the invalidation

unsigned long cur_seq

Value to pass to mmu_interval_set_seq()

Description

We temporarily evict the BO attached to this range. This necessitates evicting all user-mode queues of the process.

int **amdgpu_hmm_register**(struct amdgpu_bo *bo, unsigned long addr)

register a BO for notifier updates

Parameters

struct amdgpu_bo *bo

amdgpu buffer object

unsigned long addr

userptr addr we should monitor

Description

Registers a mmu_notifier for the given BO at the specified address. Returns 0 on success, -ERRNO if anything goes wrong.

void **amdgpu_hmm_unregister**(struct amdgpu_bo *bo)

unregister a BO for notifier updates

Parameters

struct amdgpu_bo *bo

amdgpu buffer object

Description

Remove any registration of mmu notifier updates from the buffer object.

AMDGPU Virtual Memory

GPUVM is the MMU functionality provided on the GPU. GPUVM is similar to the legacy GART on older asics, however rather than there being a single global GART table for the entire GPU, there can be multiple GPUVM page tables active at any given time. The GPUVM page tables can contain a mix VRAM pages and system pages (both memory and MMIO) and system pages can be mapped as snooped (cached system pages) or unsnooped (uncached system pages).

Each active GPUVM has an ID associated with it and there is a page table linked with each VMID. When executing a command buffer, the kernel tells the engine what VMID to use for that command buffer. VMIDs are allocated dynamically as commands are submitted. The userspace drivers maintain their own address space and the kernel sets up their pages tables accordingly when they submit their command buffers and a VMID is assigned. The hardware supports up to 16 active GPUVMs at any given time.

Each GPUVM is represented by a 1-2 or 1-5 level page table, depending on the ASIC family. GPUVM supports RWX attributes on each page as well as other features such as encryption and caching attributes.

VMID 0 is special. It is the GPUVM used for the kernel driver. In addition to an aperture managed by a page table, VMID 0 also has several other apertures. There is an aperture for direct access to VRAM and there is a legacy AGP aperture which just forwards accesses directly to the matching system physical addresses (or IOVAs when an IOMMU is present). These apertures provide direct access to these memories without incurring the overhead of a page table. VMID 0 is used by the kernel driver for tasks like memory management.

GPU clients (i.e., engines on the GPU) use GPUVM VMIDs to access memory. For user applications, each application can have their own unique GPUVM address space. The application manages the address space and the kernel driver manages the GPUVM page tables for each process. If an GPU client accesses an invalid page, it will generate a GPU page fault, similar to accessing an invalid page on a CPU.

struct **amdgpu_prt_cb**

Helper to disable partial resident texture feature from a fence callback

Definition:

```
struct amdgpu_prt_cb {
    struct amdgpu_device *adev;
    struct dma_fence_cb cb;
};
```

Members

adev

amdgpu device

cb

callback

struct **amdgpu_vm_tlb_seq_struct**

Helper to increment the TLB flush sequence

Definition:

```
struct amdgpu_vm_tlb_seq_struct {
    struct amdgpu_vm *vm;
    struct dma_fence_cb cb;
};
```

Members

vm

pointer to the amdgpu_vm structure to set the fence sequence on

cb

callback

int **amdgpu_vm_set_pasid**(struct amdgpu_device *adev, struct amdgpu_vm *vm, u32 pasid)

manage pasid and vm ptr mapping

Parameters

struct amdgpu_device *adev

amdgpu_device pointer

struct amdgpu_vm *vm

amdgpu_vm pointer

u32 pasid

the pasid the VM is using on this GPU

Description

Set the pasid this VM is using on this GPU, can also be used to remove the pasid by passing in zero.

void **amdgpu_vm_bo_evicted**(struct amdgpu_vm_bo_base *vm_bo)

vm_bo is evicted

Parameters

struct amdgpu_vm_bo_base *vm_bo

vm_bo which is evicted

Description

State for PDs/PTs and per VM BOs which are not at the location they should be.

void **amdgpu_vm_bo_moved**(struct amdgpu_vm_bo_base *vm_bo)

vm_bo is moved

Parameters

struct amdgpu_vm_bo_base *vm_bo

vm_bo which is moved

Description

State for per VM BOs which are moved, but that change is not yet reflected in the page tables.

void **amdgpu_vm_bo_idle**(struct amdgpu_vm_bo_base *vm_bo)

vm_bo is idle

Parameters

struct amdgpu_vm_bo_base *vm_bo

vm_bo which is now idle

Description

State for PDs/PTs and per VM BOs which have gone through the state machine and are now idle.

void **amdgpu_vm_bo_invalidated**(struct amdgpu_vm_bo_base *vm_bo)

vm_bo is invalidated

Parameters

struct amdgpu_vm_bo_base *vm_bo

vm_bo which is now invalidated

Description

State for normal BOs which are invalidated and that change not yet reflected in the PTs.

void **amdgpu_vm_bo_relocated**(struct amdgpu_vm_bo_base *vm_bo)
vm_bo is relocated

Parameters

struct amdgpu_vm_bo_base *vm_bo
vm_bo which is relocated

Description

State for PDs/PTs which needs to update their parent PD. For the root PD, just move to idle state.

void **amdgpu_vm_bo_done**(struct amdgpu_vm_bo_base *vm_bo)
vm_bo is done

Parameters

struct amdgpu_vm_bo_base *vm_bo
vm_bo which is now done

Description

State for normal BOs which are invalidated and that change has been updated in the PTs.

void **amdgpu_vm_bo_reset_state_machine**(struct amdgpu_vm *vm)
reset the vm_bo state machine

Parameters

struct amdgpu_vm *vm
the VM which state machine to reset

Description

Move all vm_bo object in the VM into a state where they will be updated again during validation.

void **amdgpu_vm_bo_base_init**(struct amdgpu_vm_bo_base *base, struct amdgpu_vm *vm,
struct amdgpu_bo *bo)
Adds bo to the list of bos associated with the vm

Parameters

struct amdgpu_vm_bo_base *base
base structure for tracking BO usage in a VM

struct amdgpu_vm *vm
vm to which bo is to be added

struct amdgpu_bo *bo
amdgpu buffer object

Description

Initialize a bo_va_base structure and add it to the appropriate lists

int **amdgpu_vm_lock_pd**(struct amdgpu_vm *vm, struct *drm_exec* *exec, unsigned int
num_fences)
lock PD in drm_exec

Parameters

struct amdgpu_vm *vm
vm providing the BOs

struct drm_exec *exec
drm execution context

unsigned int num_fences
number of extra fences to reserve

Description

Lock the VM root PD in the DRM execution context.

void **amdgpu_vm_move_to_lru_tail**(struct amdgpu_device *adev, struct amdgpu_vm *vm)
move all BOs to the end of LRU

Parameters

struct amdgpu_device *adev
amdgpu device pointer

struct amdgpu_vm *vm
vm providing the BOs

Description

Move all BOs to the end of LRU and remember their positions to put them together.

uint64_t **amdgpu_vm_generation**(struct amdgpu_device *adev, struct amdgpu_vm *vm)
return the page table re-generation counter

Parameters

struct amdgpu_device *adev
the amdgpu_device

struct amdgpu_vm *vm
optional VM to check, might be NULL

Description

Returns a page table re-generation token to allow checking if submissions are still valid to use this VM. The VM parameter might be NULL in which case just the VRAM lost counter will be used.

int **amdgpu_vm_validate_pt_bos**(struct amdgpu_device *adev, struct amdgpu_vm *vm, int (*validate)(void *p, struct amdgpu_bo *bo), void *param)
validate the page table BOs

Parameters

struct amdgpu_device *adev
amdgpu device pointer

struct amdgpu_vm *vm
vm providing the BOs

int (*validate)(void *p, struct amdgpu_bo *bo)
callback to do the validation

void *param
parameter for the validation callback

Description

Validate the page table BOs on command submission if neccessary.

Return

Validation result.

bool **amdgpu_vm_ready**(struct amdgpu_vm *vm)
check VM is ready for updates

Parameters

struct amdgpu_vm *vm
VM to check

Description

Check if all VM PDs/PTs are ready for updates

Return

True if VM is not evicting.

void **amdgpu_vm_check_compute_bug**(struct amdgpu_device *adev)
check whether asic has compute vm bug

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

bool **amdgpu_vm_need_pipeline_sync**(struct amdgpu_ring *ring, struct amdgpu_job *job)
Check if pipe sync is needed for job.

Parameters

struct amdgpu_ring *ring
ring on which the job will be submitted

struct amdgpu_job *job
job to submit

Return

True if sync is needed.

int **amdgpu_vm_flush**(struct amdgpu_ring *ring, struct amdgpu_job *job, bool
need_pipe_sync)
hardware flush the vm

Parameters

struct amdgpu_ring *ring
ring to use for flush

struct amdgpu_job *job
related job

bool **need_pipe_sync**
is pipe sync needed

Description

Emit a VM flush when it is necessary.

Return

0 on success, errno otherwise.

`struct amdgpu_bo_va *amdgpu_vm_bo_find(struct amdgpu_vm *vm, struct amdgpu_bo *bo)`
find the bo_va for a specific vm & bo

Parameters

`struct amdgpu_vm *vm`
requested vm

`struct amdgpu_bo *bo`
requested buffer object

Description

Find **bo** inside the requested vm. Search inside the **bos** vm list for the requested vm Returns the found bo_va or NULL if none is found

Object has to be reserved!

Return

Found bo_va or NULL.

`uint64_t amdgpu_vm_map_gart(const dma_addr_t *pages_addr, uint64_t addr)`
Resolve gart mapping of addr

Parameters

`const dma_addr_t *pages_addr`
optional DMA address to use for lookup

`uint64_t addr`
the unmapped addr

Description

Look up the physical address of the page that the pte resolves to.

Return

The pointer for the page table entry.

`int amdgpu_vm_update_pdes(struct amdgpu_device *adev, struct amdgpu_vm *vm, bool immediate)`
make sure that all directories are valid

Parameters

`struct amdgpu_device *adev`
amdgpu_device pointer

`struct amdgpu_vm *vm`
requested vm

`bool immediate`
submit immediately to the paging queue

Description

Makes sure all directories are up to date.

Return

0 for success, error for failure.

void **amdgpu_vm_tlb_seq_cb**(struct dma_fence *fence, struct dma_fence_cb *cb)
make sure to increment tlb sequence

Parameters

struct dma_fence *fence
unused

struct dma_fence_cb *cb
the callback structure

Description

Increments the tlb sequence to make sure that future CS execute a VM flush.

int **amdgpu_vm_update_range**(struct amdgpu_device *adev, struct amdgpu_vm *vm, bool immediate, bool unlocked, bool flush_tlb, struct dma_resv *resv, uint64_t start, uint64_t last, uint64_t flags, uint64_t offset, uint64_t vram_base, struct [ttm_resource](#) *res, dma_addr_t *pages_addr, struct dma_fence **fence)
update a range in the vm page table

Parameters

struct amdgpu_device *adev
amdgpu_device pointer to use for commands

struct amdgpu_vm *vm
the VM to update the range

bool immediate
immediate submission in a page fault

bool unlocked
unlocked invalidation during MM callback

bool flush_tlb
trigger tlb invalidation after update completed

struct dma_resv *resv
fences we need to sync to

uint64_t start
start of mapped range

uint64_t last
last mapped entry

uint64_t flags
flags for the entries

uint64_t offset
offset into nodes and pages_addr

uint64_t vram_base

base for vram mappings

struct ttm_resource *res

ttm_resource to map

dma_addr_t *pages_addr

DMA addresses to use for mapping

struct dma_fence **fence

optional resulting fence

Description

Fill in the page table entries between **start** and **last**.

Return

0 for success, negative error code for failure.

int **amdgpu_vm_bo_update**(struct amdgpu_device *adev, struct amdgpu_bo_va *bo_va, bool clear)

update all BO mappings in the vm page table

Parameters

struct amdgpu_device *adev

amdgpu_device pointer

struct amdgpu_bo_va *bo_va

requested BO and VM object

bool clear

if true clear the entries

Description

Fill in the page table entries for **bo_va**.

Return

0 for success, -EINVAL for failure.

void **amdgpu_vm_update_prt_state**(struct amdgpu_device *adev)

update the global PRT state

Parameters

struct amdgpu_device *adev

amdgpu_device pointer

void **amdgpu_vm_prt_get**(struct amdgpu_device *adev)

add a PRT user

Parameters

struct amdgpu_device *adev

amdgpu_device pointer

void **amdgpu_vm_prt_put**(struct amdgpu_device *adev)

drop a PRT user

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

void **amdgpu_vm_prt_cb**(struct dma_fence *fence, struct dma_fence_cb *_cb)
callback for updating the PRT status

Parameters

struct dma_fence *fence
fence for the callback

struct dma_fence_cb *_cb
the callback function

void **amdgpu_vm_add_prt_cb**(struct amdgpu_device *adev, struct dma_fence *fence)
add callback for updating the PRT status

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct dma_fence *fence
fence for the callback

void **amdgpu_vm_free_mapping**(struct amdgpu_device *adev, struct amdgpu_vm *vm, struct
amdgpu_bo_va_mapping *mapping, struct dma_fence *fence)
free a mapping

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
requested vm

struct amdgpu_bo_va_mapping *mapping
mapping to be freed

struct dma_fence *fence
fence of the unmap operation

Description

Free a mapping and make sure we decrease the PRT usage count if applicable.

void **amdgpu_vm_prt_fini**(struct amdgpu_device *adev, struct amdgpu_vm *vm)
finish all prt mappings

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
requested vm

Description

Register a cleanup callback to disable PRT support after VM dies.

```
int amdgpu_vm_clear_freed(struct amdgpu_device *adev, struct amdgpu_vm *vm, struct
                        dma_fence **fence)
```

clear freed BOs in the PT

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
requested vm

struct dma_fence **fence
optional resulting fence (unchanged if no work needed to be done or if an error occurred)

Description

Make sure all freed BOs are cleared in the PT. PTs have to be reserved and mutex must be locked!

Return

0 for success.

```
int amdgpu_vm_handle_moved(struct amdgpu_device *adev, struct amdgpu_vm *vm)
    handle moved BOs in the PT
```

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
requested vm

Description

Make sure all BOs which are moved are updated in the PTs.

PTs have to be reserved!

Return

0 for success.

```
struct amdgpu_bo_va *amdgpu_vm_bo_add(struct amdgpu_device *adev, struct amdgpu_vm
                                      *vm, struct amdgpu_bo *bo)
```

add a bo to a specific vm

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
requested vm

struct amdgpu_bo *bo
amdgpu buffer object

Description

Add **bo** into the requested vm. Add **bo** to the list of bos associated with the vm

Object has to be reserved!

Return

Newly added bo_va or NULL for failure

```
void amdgpu_vm_bo_insert_map(struct amdgpu_device *adev, struct amdgpu_bo_va *bo_va,  
                             struct amdgpu_bo_va_mapping *mapping)
```

insert a new mapping

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_bo_va *bo_va
bo_va to store the address

struct amdgpu_bo_va_mapping *mapping
the mapping to insert

Description

Insert a new mapping into all structures.

```
int amdgpu_vm_bo_map(struct amdgpu_device *adev, struct amdgpu_bo_va *bo_va, uint64_t  
                     saddr, uint64_t offset, uint64_t size, uint64_t flags)
```

map bo inside a vm

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_bo_va *bo_va
bo_va to store the address

uint64_t saddr
where to map the BO

uint64_t offset
requested offset in the BO

uint64_t size
BO size in bytes

uint64_t flags
attributes of pages (read/write/valid/etc.)

Description

Add a mapping of the BO at the specefied addr into the VM.

Object has to be reserved and unreserved outside!

Return

0 for success, error for failure.

int **amdgpu_vm_bo_replace_map**(struct amdgpu_device *adev, struct amdgpu_bo_va *bo_va,
uint64_t saddr, uint64_t offset, uint64_t size, uint64_t flags)

map bo inside a vm, replacing existing mappings

Parameters

struct amdgpu_device *adev

amdgpu_device pointer

struct amdgpu_bo_va *bo_va

bo_va to store the address

uint64_t saddr

where to map the BO

uint64_t offset

requested offset in the BO

uint64_t size

BO size in bytes

uint64_t flags

attributes of pages (read/write/valid/etc.)

Description

Add a mapping of the BO at the specefied addr into the VM. Replace existing mappings as we do so.

Object has to be reserved and unreserved outside!

Return

0 for success, error for failure.

int **amdgpu_vm_bo_unmap**(struct amdgpu_device *adev, struct amdgpu_bo_va *bo_va, uint64_t
saddr)

remove bo mapping from vm

Parameters

struct amdgpu_device *adev

amdgpu_device pointer

struct amdgpu_bo_va *bo_va

bo_va to remove the address from

uint64_t saddr

where to the BO is mapped

Description

Remove a mapping of the BO at the specefied addr from the VM.

Object has to be reserved and unreserved outside!

Return

0 for success, error for failure.

int **amdgpu_vm_bo_clear_mappings**(struct amdgpu_device *adev, struct amdgpu_vm *vm,
uint64_t saddr, uint64_t size)

remove all mappings in a specific range

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
VM structure to use

uint64_t saddr
start of the range

uint64_t size
size of the range

Description

Remove all mappings in a range, split them as appropriate.

Return

0 for success, error for failure.

struct amdgpu_bo_va_mapping ***amdgpu_vm_bo_lookup_mapping**(struct amdgpu_vm *vm,
uint64_t addr)

find mapping by address

Parameters

struct amdgpu_vm *vm
the requested VM

uint64_t addr
the address

Description

Find a mapping by it's address.

Return

The amdgpu_bo_va_mapping matching for addr or NULL

void **amdgpu_vm_bo_trace_cs**(struct amdgpu_vm *vm, struct ww_acquire_ctx *ticket)
trace all reserved mappings

Parameters

struct amdgpu_vm *vm
the requested vm

struct ww_acquire_ctx *ticket
CS ticket

Description

Trace all mappings of BOs reserved during a command submission.

void **amdgpu_vm_bo_del**(struct amdgpu_device *adev, struct amdgpu_bo_va *bo_va)
remove a bo from a specific vm

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_bo_va *bo_va
requested bo_va

Description

Remove **bo_va->bo** from the requested vm.

Object have to be reserved!

bool **amdgpu_vm_evictable**(struct amdgpu_bo *bo)
check if we can evict a VM

Parameters

struct amdgpu_bo *bo
A page table of the VM.

Description

Check if it is possible to evict a VM.

void **amdgpu_vm_bo_invalidate**(struct amdgpu_device *adev, struct amdgpu_bo *bo, bool evicted)
mark the bo as invalid

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_bo *bo
amdgpu buffer object

bool evicted
is the BO evicted

Description

Mark **bo** as invalid.

uint32_t **amdgpu_vm_get_block_size**(uint64_t vm_size)
calculate VM page table size as power of two

Parameters

uint64_t vm_size
VM size

Return

VM page table as power of two

void **amdgpu_vm_adjust_size**(struct amdgpu_device *adev, uint32_t min_vm_size, uint32_t fragment_size_default, unsigned max_level, unsigned max_bits)

adjust vm size, block size and fragment size

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

uint32_t min_vm_size
the minimum vm size in GB if it's set auto

uint32_t fragment_size_default
Default PTE fragment size

unsigned max_level
max VMPT level

unsigned max_bits
max address space size in bits

long **amdgpu_vm_wait_idle**(struct amdgpu_vm *vm, long timeout)
wait for the VM to become idle

Parameters

struct amdgpu_vm *vm
VM object to wait for

long timeout
timeout to wait for VM to become idle

int **amdgpu_vm_init**(struct amdgpu_device *adev, struct amdgpu_vm *vm, int32_t xcp_id)
initialize a vm instance

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
requested vm

int32_t xcp_id
GPU partition selection id

Description

Init **vm** fields.

Return

0 for success, error for failure.

int **amdgpu_vm_make_compute**(struct amdgpu_device *adev, struct amdgpu_vm *vm)
Turn a GFX VM into a compute VM

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
requested vm

Description

This only works on GFX VMs that don't have any BOs added and no page tables allocated yet. Changes the following VM parameters: - use_cpu_for_update - pte_supports_ats
Reinitializes the page directory to reflect the changed ATS setting.

Return

0 for success, -errno for errors.

void **amdgpu_vm_release_compute**(struct amdgpu_device *adev, struct amdgpu_vm *vm)
release a compute vm

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
a vm turned into compute vm by calling amdgpu_vm_make_compute

Description

This is a correspondent of amdgpu_vm_make_compute. It decouples compute pasid from vm. Compute should stop use of vm after this call.

void **amdgpu_vm_fini**(struct amdgpu_device *adev, struct amdgpu_vm *vm)
tear down a vm instance

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

struct amdgpu_vm *vm
requested vm

Description

Tear down **vm**. Unbind the VM and remove all bos from the vm bo list

void **amdgpu_vm_manager_init**(struct amdgpu_device *adev)
init the VM manager

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

Description

Initialize the VM manager structures

void **amdgpu_vm_manager_fini**(struct amdgpu_device *adev)
cleanup VM manager

Parameters

struct amdgpu_device *adev
amdgpu_device pointer

Description

Cleanup the VM manager and free resources.

int **amdgpu_vm_ioctl**(struct *drm_device* *dev, void *data, struct *drm_file* *filp)
Manages VMID reservation for vm hubs.

Parameters

struct drm_device *dev
drm device pointer

void *data
drm_amdgpu_vm

struct drm_file *filp
drm file pointer

Return

0 for success, -errno for errors.

void **amdgpu_vm_get_task_info**(struct amdgpu_device *adev, u32 pasid, struct
amdgpu_task_info *task_info)
Extracts task info for a PASID.

Parameters

struct amdgpu_device *adev
drm device pointer

u32 pasid
PASID identifier for VM

struct amdgpu_task_info *task_info
task_info to fill.

void **amdgpu_vm_set_task_info**(struct amdgpu_vm *vm)
Sets VMs task info.

Parameters

struct amdgpu_vm *vm
vm for which to set the info

bool **amdgpu_vm_handle_fault**(struct amdgpu_device *adev, u32 pasid, u32 vmid, u32
node_id, uint64_t addr, bool write_fault)
graceful handling of VM faults.

Parameters

struct amdgpu_device *adev
amdgpu device pointer

u32 pasid

PASID of the VM

u32 vmid

VMID, only used for GFX 9.4.3.

u32 node_id

Node_id received in IH cookie. Only applicable for GFX 9.4.3.

uint64_t addr

Address of the fault

bool write_fault

true is write fault, false is read fault

Description

Try to gracefully handle a VM fault. Return true if the fault was handled and shouldn't be reported any more.

```
void amdgpu_debugfs_vm_bo_info(struct amdgpu_vm *vm, struct seq_file *m)
    print BO info for the VM
```

Parameters

```
struct amdgpu_vm *vm
```

Requested VM for printing BO info

```
struct seq_file *m
```

debugfs file

Description

Print BO information in debugfs file for the VM

Interrupt Handling

Interrupts generated within GPU hardware raise interrupt requests that are passed to amdgpu IRQ handler which is responsible for detecting source and type of the interrupt and dispatching matching handlers. If handling an interrupt requires calling kernel functions that may sleep processing is dispatched to work handlers.

If MSI functionality is not disabled by module parameter then MSI support will be enabled.

For GPU interrupt sources that may be driven by another driver, IRQ domain support is used (with mapping between virtual and hardware IRQs).

```
void amdgpu_irq_disable_all(struct amdgpu_device *adev)
    disable all interrupts
```

Parameters

```
struct amdgpu_device *adev
```

amdgpu device pointer

Description

Disable all types of interrupts from all sources.

`irqreturn_t amdgpu_irq_handler(int irq, void *arg)`
IRQ handler

Parameters

int irq
IRQ number (unused)

void *arg
pointer to DRM device

Description

IRQ handler for amdgpu driver (all ASICs).

Return

result of handling the IRQ, as defined by `irqreturn_t`

`void amdgpu_irq_handle_ih1(struct work_struct *work)`
kick of processing for IH1

Parameters

struct work_struct *work
work structure in `struct amdgpu_irq`

Description

Kick of processing IH ring 1.

`void amdgpu_irq_handle_ih2(struct work_struct *work)`
kick of processing for IH2

Parameters

struct work_struct *work
work structure in `struct amdgpu_irq`

Description

Kick of processing IH ring 2.

`void amdgpu_irq_handle_ih_soft(struct work_struct *work)`
kick of processing for `ih_soft`

Parameters

struct work_struct *work
work structure in `struct amdgpu_irq`

Description

Kick of processing IH soft ring.

`bool amdgpu_msi_ok(struct amdgpu_device *adev)`
check whether MSI functionality is enabled

Parameters

struct amdgpu_device *adev
amdgpu device pointer (unused)

Description

Checks whether MSI functionality has been disabled via module parameter (all ASICs).

Return

true if MSIs are allowed to be enabled or *false* otherwise

int **amdgpu_irq_init**(struct amdgpu_device *adev)
initialize interrupt handling

Parameters

struct amdgpu_device *adev
amdgpu device pointer

Description

Sets up work functions for hotplug and reset interrupts, enables MSI functionality, initializes vblank, hotplug and reset interrupt handling.

Return

0 on success or error code on failure

void **amdgpu_irq_fini_sw**(struct amdgpu_device *adev)
shut down interrupt handling

Parameters

struct amdgpu_device *adev
amdgpu device pointer

Description

Tears down work functions for hotplug and reset interrupts, disables MSI functionality, shuts down vblank, hotplug and reset interrupt handling, turns off interrupts from all sources (all ASICs).

int **amdgpu_irq_add_id**(struct amdgpu_device *adev, unsigned int client_id, unsigned int src_id, struct amdgpu_irq_src *source)
register IRQ source

Parameters

struct amdgpu_device *adev
amdgpu device pointer

unsigned int client_id
client id

unsigned int src_id
source id

struct amdgpu_irq_src *source
IRQ source pointer

Description

Registers IRQ source on a client.

Return

0 on success or error code otherwise

void **amdgpu_irq_dispatch**(struct amdgpu_device *adev, struct amdgpu_ih_ring *ih)
dispatch IRQ to IP blocks

Parameters

struct amdgpu_device *adev
amdgpu device pointer

struct amdgpu_ih_ring *ih
interrupt ring instance

Description

Dispatches IRQ to IP blocks.

void **amdgpu_irq_delegate**(struct amdgpu_device *adev, struct amdgpu_iv_entry *entry,
unsigned int num_dw)
delegate IV to soft IH ring

Parameters

struct amdgpu_device *adev
amdgpu device pointer

struct amdgpu_iv_entry *entry
IV entry

unsigned int num_dw
size of IV

Description

Delegate the IV to the soft IH ring and schedule processing of it. Used if the hardware delegation to IH1 or IH2 doesn't work for some reason.

int **amdgpu_irq_update**(struct amdgpu_device *adev, struct amdgpu_irq_src *src, unsigned
int type)
update hardware interrupt state

Parameters

struct amdgpu_device *adev
amdgpu device pointer

struct amdgpu_irq_src *src
interrupt source pointer

unsigned int type
type of interrupt

Description

Updates interrupt state for the specific source (all ASICs).

void **amdgpu_irq_gpu_reset_resume_helper**(struct amdgpu_device *adev)
update interrupt states on all sources

Parameters

struct amdgpu_device *adev
amdgpu device pointer

Description

Updates state of all types of interrupts on all sources on resume after reset.

int **amdgpu_irq_get**(struct amdgpu_device *adev, struct amdgpu_irq_src *src, unsigned int type)
enable interrupt

Parameters

struct amdgpu_device *adev
amdgpu device pointer

struct amdgpu_irq_src *src
interrupt source pointer

unsigned int type
type of interrupt

Description

Enables specified type of interrupt on the specified source (all ASICs).

Return

0 on success or error code otherwise

int **amdgpu_irq_put**(struct amdgpu_device *adev, struct amdgpu_irq_src *src, unsigned int type)
disable interrupt

Parameters

struct amdgpu_device *adev
amdgpu device pointer

struct amdgpu_irq_src *src
interrupt source pointer

unsigned int type
type of interrupt

Description

Enables specified type of interrupt on the specified source (all ASICs).

Return

0 on success or error code otherwise

bool **amdgpu_irq_enabled**(struct amdgpu_device *adev, struct amdgpu_irq_src *src, unsigned int type)
check whether interrupt is enabled or not

Parameters

struct amdgpu_device *adev
amdgpu device pointer

struct amdgpu_irq_src *src
interrupt source pointer

unsigned int type
type of interrupt

Description

Checks whether the given type of interrupt is enabled on the given source.

Return

true if interrupt is enabled, *false* if interrupt is disabled or on invalid parameters

int **amdgpu_irqdomain_map**(struct irq_domain *d, unsigned int irq, irq_hw_number_t hwirq)
create mapping between virtual and hardware IRQ numbers

Parameters

struct irq_domain *d
amdgpu IRQ domain pointer (unused)

unsigned int irq
virtual IRQ number

irq_hw_number_t hwirq
hardware irq number

Description

Current implementation assigns simple interrupt handler to the given virtual IRQ.

Return

0 on success or error code otherwise

int **amdgpu_irq_add_domain**(struct amdgpu_device *adev)
create a linear IRQ domain

Parameters

struct amdgpu_device *adev
amdgpu device pointer

Description

Creates an IRQ domain for GPU interrupt sources that may be driven by another driver (e.g., ACP).

Return

0 on success or error code otherwise

void **amdgpu_irq_remove_domain**(struct amdgpu_device *adev)
remove the IRQ domain

Parameters

struct amdgpu_device *adev
amdgpu device pointer

Description

Removes the IRQ domain for GPU interrupt sources that may be driven by another driver (e.g., ACP).

unsigned int **amdgpu_irq_create_mapping**(struct amdgpu_device *adev, unsigned int src_id)
create mapping between domain Linux IRQs

Parameters

struct amdgpu_device *adev
amdgpu device pointer

unsigned int src_id
IH source id

Description

Creates mapping between a domain IRQ (GPU IH src id) and a Linux IRQ Use this for components that generate a GPU interrupt, but are driven by a different driver (e.g., ACP).

Return

Linux IRQ

IP Blocks

GPUs are composed of IP (intellectual property) blocks. These IP blocks provide various functionalities: display, graphics, video decode, etc. The IP blocks that comprise a particular GPU are listed in the GPU's respective SoC file. `amdgpu_device.c` acquires the list of IP blocks for the GPU in use on initialization. It can then operate on this list to perform standard driver operations such as: init, fini, suspend, resume, etc.

IP block implementations are named using the following convention: `<functionality>_v<version>` (E.g.: `gfx_v6_0`).

enum **amd_ip_block_type**
Used to classify IP blocks by functionality.

Constants

AMD_IP_BLOCK_TYPE_COMMON
GPU Family

AMD_IP_BLOCK_TYPE_GMC
Graphics Memory Controller

AMD_IP_BLOCK_TYPE_IH
Interrupt Handler

AMD_IP_BLOCK_TYPE_SMC
System Management Controller

AMD_IP_BLOCK_TYPE_PSP
Platform Security Processor

AMD_IP_BLOCK_TYPE_DCE
Display and Compositing Engine

AMD_IP_BLOCK_TYPE_GFX

Graphics and Compute Engine

AMD_IP_BLOCK_TYPE_SDMA

System DMA Engine

AMD_IP_BLOCK_TYPE_UVD

Unified Video Decoder

AMD_IP_BLOCK_TYPE_VCE

Video Compression Engine

AMD_IP_BLOCK_TYPE_ACP

Audio Co-Processor

AMD_IP_BLOCK_TYPE_VCN

Video Core/Codec Next

AMD_IP_BLOCK_TYPE_MES

Micro-Engine Scheduler

AMD_IP_BLOCK_TYPE_JPEG

JPEG Engine

AMD_IP_BLOCK_TYPE_NUM

Total number of IP block types

struct **amd_ip_funcs**

general hooks for managing amdgpu IP Blocks

Definition:

```

struct amd_ip_funcs {
    char *name;
    int (*early_init)(void *handle);
    int (*late_init)(void *handle);
    int (*sw_init)(void *handle);
    int (*sw_fini)(void *handle);
    int (*early_fini)(void *handle);
    int (*hw_init)(void *handle);
    int (*hw_fini)(void *handle);
    void (*late_fini)(void *handle);
    int (*prepare_suspend)(void *handle);
    int (*suspend)(void *handle);
    int (*resume)(void *handle);
    bool (*is_idle)(void *handle);
    int (*wait_for_idle)(void *handle);
    bool (*check_soft_reset)(void *handle);
    int (*pre_soft_reset)(void *handle);
    int (*soft_reset)(void *handle);
    int (*post_soft_reset)(void *handle);
    int (*set_clockgating_state)(void *handle, enum amd_clockgating_state,
↪state);
    int (*set_powergating_state)(void *handle, enum amd_powergating_state,
↪state);

```

```
void (*get_clockgating_state)(void *handle, u64 *flags);  
};
```

Members**name**

Name of IP block

early_init

sets up early driver state (pre sw_init), does not configure hw - Optional

late_init

sets up late driver/hw state (post hw_init) - Optional

sw_init

sets up driver state, does not configure hw

sw_fini

tears down driver state, does not configure hw

early_fini

tears down stuff before dev detached from driver

hw_init

sets up the hw state

hw_fini

tears down the hw state

late_fini

final cleanup

suspend

handles IP specific hw/sw changes for suspend

resume

handles IP specific hw/sw changes for resume

is_idle

returns current IP block idle status

wait_for_idle

poll for idle

check_soft_reset

check soft reset the IP block

pre_soft_reset

pre soft reset the IP block

soft_reset

soft reset the IP block

post_soft_reset

post soft reset the IP block

set_clockgating_state

enable/disable cg for the IP block

set_powergating_state

enable/disable pg for the IP block

get_clockgating_state

get current clockgating status

Description

These hooks provide an interface for controlling the operational state of IP blocks. After acquiring a list of IP blocks for the GPU in use, the driver can make chip-wide state changes by walking this list and making calls to hooks from each IP block. This list is ordered to ensure that the driver initializes the IP blocks in a safe sequence.

10.1.3 drm/amd/display - Display Core (DC)

AMD display engine is partially shared with other operating systems; for this reason, our Display Core Driver is divided into two pieces:

1. **Display Core (DC)** contains the OS-agnostic components. Things like hardware programming and resource management are handled here.
2. **Display Manager (DM)** contains the OS-dependent components. Hooks to the amdgpu base driver and DRM are implemented here.

The display pipe is responsible for “scanning out” a rendered frame from the GPU memory (also called VRAM, FrameBuffer, etc.) to a display. In other words, it would:

1. Read frame information from memory;
2. Perform required transformation;
3. Send pixel data to sink devices.

If you want to learn more about our driver details, take a look at the below table of content:

AMDgpu Display Manager**Table of Contents**

- *AMDgpu Display Manager*
 - *Lifecycle*
 - *Interrupts*
 - *Atomic Implementation*
 - *Color Management Properties*
 - * *DC Color Capabilities between DCN generations*
 - *Blend Mode Properties*
 - * *Blend configuration flow*

The AMDgpu display manager, **amdgpu_dm** (or even simpler, **dm**) sits between DRM and DC. It acts as a liaison, converting DRM requests into DC requests, and DC responses into DRM responses.

The root control structure is *struct amdgpu_display_manager*.

struct **dm_compressor_info**

Buffer info used by frame buffer compression

Definition:

```
struct dm_compressor_info {  
    void *cpu_addr;  
    struct amdgpu_bo *bo_ptr;  
    uint64_t gpu_addr;  
};
```

Members

cpu_addr

MMIO cpu addr

bo_ptr

Pointer to the buffer object

gpu_addr

MMIO gpu addr

struct **dmub_hpd_work**

Handle time consuming work in low priority outbox IRQ

Definition:

```
struct dmub_hpd_work {  
    struct work_struct handle_hpd_work;  
    struct dmub_notification *dmub_notify;  
    struct amdgpu_device *adev;  
};
```

Members

handle_hpd_work

Work to be executed in a separate thread to handle hpd_low_irq

dmub_notify

notification for callback function

adev

amdgpu_device pointer

struct **vblank_control_work**

Work data for vblank control

Definition:

```
struct vblank_control_work {  
    struct work_struct work;  
    struct amdgpu_display_manager *dm;
```

```
struct amdgpu_crtc *acrtc;
struct dc_stream_state *stream;
bool enable;
};
```

Members

work

Kernel work data for the work event

dm

amdgpu display manager device

acrtc

amdgpu CRTC instance for which the event has occurred

stream

DC stream for which the event has occurred

enable

true if enabling vblank

struct **amdgpu_dm_backlight_caps**

Information about backlight

Definition:

```
struct amdgpu_dm_backlight_caps {
    union dpcd_sink_ext_caps *ext_caps;
    u32 aux_min_input_signal;
    u32 aux_max_input_signal;
    int min_input_signal;
    int max_input_signal;
    bool caps_valid;
    bool aux_support;
};
```

Members

ext_caps

Keep the data struct with all the information about the display support for HDR.

aux_min_input_signal

Min brightness value supported by the display

aux_max_input_signal

Max brightness value supported by the display in nits.

min_input_signal

minimum possible input in range 0-255.

max_input_signal

maximum possible input in range 0-255.

caps_valid

true if these values are from the ACPI interface.

aux_support

Describes if the display supports AUX backlight.

Description

Describe the backlight support for ACPI or eDP AUX.

struct dal_allocation

Tracks mapped FB memory for SMU communication

Definition:

```
struct dal_allocation {
    struct list_head list;
    struct amdgpu_bo *bo;
    void *cpu_ptr;
    u64 gpu_addr;
};
```

Members

list

list of dal allocations

bo

GPU buffer object

cpu_ptr

CPU virtual address of the GPU buffer object

gpu_addr

GPU virtual address of the GPU buffer object

struct hpd_rx_irq_offload_work_queue

Work queue to handle hpd_rx_irq offload work

Definition:

```
struct hpd_rx_irq_offload_work_queue {
    struct workqueue_struct *wq;
    spinlock_t offload_lock;
    bool is_handling_link_loss;
    bool is_handling_mst_msg_rdy_event;
    struct amdgpu_dm_connector *aconnector;
};
```

Members

wq

workqueue structure to queue offload work.

offload_lock

To protect fields of offload work queue.

is_handling_link_loss

Used to prevent inserting link loss event when we're handling link loss

is_handling_mst_msg_rdy_event

Used to prevent inserting mst message ready event when we're already handling mst message ready event

aconnector

The aconnector that this work queue is attached to

struct hpd_rx_irq_offload_work

hpd_rx_irq offload work structure

Definition:

```
struct hpd_rx_irq_offload_work {
    struct work_struct work;
    union hpd_irq_data data;
    struct hpd_rx_irq_offload_work_queue *offload_wq;
};
```

Members**work**

offload work

data

reference irq data which is used while handling offload work

offload_wq

offload work queue that this work is queued to

struct amdgpu_display_manager

Central amdgpu display manager device

Definition:

```
struct amdgpu_display_manager {
    struct dc *dc;
    struct dmub_srv *dmub_srv;
    struct dmub_notification *dmub_notify;
    dmub_notify_interrupt_callback_t dmub_callback[AMDGPU_DMUB_NOTIFICATION_
→MAX];
    bool dmub_thread_offload[AMDGPU_DMUB_NOTIFICATION_MAX];
    struct dmub_srv_fb_info *dmub_fb_info;
    const struct firmware *dmub_fw;
    struct amdgpu_bo *dmub_bo;
    u64 dmub_bo_gpu_addr;
    void *dmub_bo_cpu_addr;
    uint32_t dmcub_fw_version;
    struct cgs_device *cgs_device;
    struct amdgpu_device *adev;
    struct drm_device *ddev;
    u16 display_indexes_num;
    struct drm_private_obj atomic_obj;
    struct mutex dc_lock;
    struct mutex audio_lock;
    struct drm_audio_component *audio_component;
};
```

```

bool audio_registered;
struct list_head irq_handler_list_low_tab[DAL_IRQ_SOURCES_NUMBER];
struct list_head irq_handler_list_high_tab[DAL_IRQ_SOURCES_NUMBER];
struct common_irq_params pflip_params[DC_IRQ_SOURCE_PFLIP_LAST - DC_IRQ_
→SOURCE_PFLIP_FIRST + 1];
struct common_irq_params vblank_params[DC_IRQ_SOURCE_VBLANK6 - DC_IRQ_
→SOURCE_VBLANK1 + 1];
struct common_irq_params vline0_params[DC_IRQ_SOURCE_DC6_VLINE0 - DC_IRQ_
→SOURCE_DC1_VLINE0 + 1];
struct common_irq_params vupdate_params[DC_IRQ_SOURCE_VUPDATE6 - DC_IRQ_
→SOURCE_VUPDATE1 + 1];
struct common_irq_params dmub_trace_params[1];
struct common_irq_params dmub_outbox_params[1];
spinlock_t irq_handler_list_table_lock;
struct backlight_device *backlight_dev[AMDGPU_DM_MAX_NUM_EDP];
const struct dc_link *backlight_link[AMDGPU_DM_MAX_NUM_EDP];
uint8_t num_of_edps;
struct amdgpu_dm_backlight_caps backlight_caps[AMDGPU_DM_MAX_NUM_EDP];
struct mod_freesync *freesync_module;
struct hdcp_workqueue *hdcp_workqueue;
struct workqueue_struct *vblank_control_workqueue;
struct drm_atomic_state *cached_state;
struct dc_state *cached_dc_state;
struct dm_compressor_info compressor;
const struct firmware *fw_dmcu;
uint32_t dmcu_fw_version;
const struct gpu_info_soc_bounding_box_v1_0 *soc_bounding_box;
uint32_t active_vblank_irq_count;
#ifdef CONFIG_DRM_AMD_SECURE_DISPLAY;
struct secure_display_context *secure_display_ctxs;
#endif;
struct hpd_rx_irq_offload_work_queue *hpd_rx_offload_wq;
struct amdgpu_encoder mst_encoders[AMDGPU_DM_MAX_CRTC];
bool force_timing_sync;
bool disable_hpd_irq;
bool dmcub_trace_event_en;
struct list_head da_list;
struct completion dmub_aux_transfer_done;
struct workqueue_struct *delayed_hpd_wq;
u32 brightness[AMDGPU_DM_MAX_NUM_EDP];
u32 actual_brightness[AMDGPU_DM_MAX_NUM_EDP];
bool aux_hpd_discon_quirk;
struct mutex dpia_aux_lock;
};

```

Members

dc

Display Core control structure

dmub_srv

DMUB service, used for controlling the DMUB on hardware that supports it. The pointer

to the `dmub_srv` will be `NULL` on hardware that does not support it.

dmub_notify

Notification from DMUB.

dmub_callback

Callback functions to handle notification from DMUB.

dmub_thread_offload

Flag to indicate if callback is offload.

dmub_fb_info

Framebuffer regions for the DMUB.

dmub_fw

DMUB firmware, required on hardware that has DMUB support.

dmub_bo

Buffer object for the DMUB.

dmub_bo_gpu_addr

GPU virtual address for the DMUB buffer object.

dmub_bo_cpu_addr

CPU address for the DMUB buffer object.

dmcub_fw_version

DMCUB firmware version.

cgs_device

The Common Graphics Services device. It provides an interface for accessing registers.

adev

AMDGPU base driver structure

ddev

DRM base driver structure

display_indexes_num

Max number of display streams supported

atomic_obj

In combination with `dm_atomic_state` it helps manage global atomic state that doesn't map cleanly into existing drm resources, like `dc_context`.

dc_lock

Guards access to DC functions that can issue register write sequences.

audio_lock

Guards access to audio instance changes.

audio_component

Used to notify ELD changes to sound driver.

audio_registered

True if the audio component has been registered successfully, false otherwise.

irq_handler_list_low_tab

Low priority IRQ handler table.

It is a $n \times m$ table consisting of n IRQ sources, and m handlers per IRQ source. Low priority IRQ handlers are deferred to a workqueue to be processed. Hence, they can sleep.

Note that handlers are called in the same order as they were registered (FIFO).

irq_handler_list_high_tab

High priority IRQ handler table.

It is a $n \times m$ table, same as `irq_handler_list_low_tab`. However, handlers in this table are not deferred and are called immediately.

pflip_params

Page flip IRQ parameters, passed to registered handlers when triggered.

vblank_params

Vertical blanking IRQ parameters, passed to registered handlers when triggered.

vline0_params

OTG vertical interrupt0 IRQ parameters, passed to registered handlers when triggered.

vupdate_params

Vertical update IRQ parameters, passed to registered handlers when triggered.

dmub_trace_params

DMUB trace event IRQ parameters, passed to registered handlers when triggered.

dmub_outbox_params

DMUB Outbox parameters

irq_handler_list_table_lock

Synchronizes access to IRQ tables

backlight_dev

Backlight control device

backlight_link

Link on which to control backlight

num_of_edps

number of backlight eDPs

backlight_caps

Capabilities of the backlight device

freesync_module

Module handling freesync calculations

hdcp_workqueue

AMDGPU content protection queue

vblank_control_workqueue

Deferred work for vblank control events.

cached_state

Caches device atomic state for suspend/resume

cached_dc_state

Cached state of content streams

compressor

Frame buffer compression buffer. See [*struct dm_compressor_info*](#)

fw_dmcu

Reference to DMCU firmware

dmcu_fw_version

Version of the DMCU firmware

soc_bounding_box

gpu_info FW provided soc bounding box struct or 0 if not available in FW

active_vblank_irq_count

number of currently active vblank irqs

secure_display_ctxs

Store the ROI information and the work_struct to command dmub and psp for all crtcs.

hpd_rx_offload_wq

Work queue to offload works of hpd_rx_irq

mst_encoders

fake encoders used for DP MST.

force_timing_sync

set via debugfs. When set, indicates that all connected displays will be forced to synchronize.

disable_hpd_irq

disables all HPD and HPD RX interrupt handling in the driver when true

dmub_trace_event_en

enable dmub trace events

da_list

DAL fb memory allocation list, for communication with SMU.

dmub_aux_transfer_done

struct completion used to indicate when DMUB transfers are done

delayed_hpd_wq

work queue used to delay DMUB HPD work

brightness

cached backlight values.

actual_brightness

last successfully applied backlight values.

aux_hpd_discon_quirk

quirk for hpd discon while aux is on-going. occurred on certain intel platform

dpia_aux_lock

Guards access to DPIA AUX

struct **amdgpu_hdmi_vsdb_info**

Keep track of the VSDB info

Definition:

```
struct amdgpu_hdmi_vsdb_info {
    unsigned int amd_vsdb_version;
    bool freesync_supported;
```

```
    unsigned int min_refresh_rate_hz;
    unsigned int max_refresh_rate_hz;
    bool replay_mode;
};
```

Members

amd_vsdb_version

Vendor Specific Data Block Version, should be used to determine which Vendor Specific InfoFrame (VSIF) to send.

freesync_supported

FreeSync Supported.

min_refresh_rate_hz

FreeSync Minimum Refresh Rate in Hz.

max_refresh_rate_hz

FreeSync Maximum Refresh Rate in Hz

replay_mode

Replay supported

Description

AMDGPU supports FreeSync over HDMI by using the VSDB section, and this struct is useful to keep track of the display-specific information about FreeSync.

Lifecycle

DM (and consequently DC) is registered in the amdgpu base driver as a IP block. When CONFIG_DRM_AMD_DC is enabled, the DM device IP block is added to the base driver's device list to be initialized and torn down accordingly.

The functions to do so are provided as hooks in *struct amd_ip_funcs*.

int **dm_hw_init**(void *handle)

Initialize DC device

Parameters

void *handle

The base driver device containing the amdgpu_dm device.

Description

Initialize the *struct amdgpu_display_manager* device. This involves calling the initializers of each DM component, then populating the struct with them.

Although the function implies hardware initialization, both hardware and software are initialized here. Splitting them out to their relevant init hooks is a future TODO item.

Some notable things that are initialized here:

- Display Core, both software and hardware
- DC modules that we need (freesync and color management)
- DRM software states

- Interrupt sources and handlers
- Vblank support
- Debug FS entries, if enabled

int **dm_hw_fini**(void *handle)
Teardown DC device

Parameters

void *handle

The base driver device containing the amdgpu_dm device.

Description

Teardown components within *struct amdgpu_display_manager* that require cleanup. This involves cleaning up the DRM device, DC, and any modules that were loaded. Also flush IRQ workqueues and disable them.

Interrupts

DM provides another layer of IRQ management on top of what the base driver already provides. This is something that could be cleaned up, and is a future TODO item.

The base driver provides IRQ source registration with DRM, handler registration into the base driver's IRQ table, and a handler callback *amdgpu_irq_handler()*, with which DRM calls on interrupts. This generic handler looks up the IRQ table, and calls the respective *amdgpu_irq_src_funcs.process* hookups.

What DM provides on top are two IRQ tables specifically for top-half and bottom-half IRQ handling, with the bottom-half implementing workqueues:

- *amdgpu_display_manager.irq_handler_list_high_tab*
- *amdgpu_display_manager.irq_handler_list_low_tab*

They override the base driver's IRQ table, and the effect can be seen in the hooks that DM provides for *amdgpu_irq_src_funcs.process*. They are all set to the DM generic handler *amdgpu_dm_irq_handler()*, which looks up DM's IRQ tables. However, in order for base driver to recognize this hook, DM still needs to register the IRQ with the base driver. See *dce110_register_irq_handlers()* and *dcn10_register_irq_handlers()*.

To expose DC's hardware interrupt toggle to the base driver, DM implements *amdgpu_irq_src_funcs.set* hooks. Base driver calls it through *amdgpu_irq_update()* to enable or disable the interrupt.

struct **amdgpu_dm_irq_handler_data**
Data for DM interrupt handlers.

Definition:

```
struct amdgpu_dm_irq_handler_data {
    struct list_head list;
    interrupt_handler handler;
    void *handler_arg;
    struct amdgpu_display_manager *dm;
```

```
enum dc_irq_source irq_source;
struct work_struct work;
};
```

Members

list

Linked list entry referencing the next/previous handler

handler

Handler function

handler_arg

Argument passed to the handler when triggered

dm

DM which this handler belongs to

irq_source

DC interrupt source that this handler is registered for

work

work struct

void **dm_irq_work_func**(struct work_struct *work)

Handle an IRQ outside of the interrupt handler proper.

Parameters

struct work_struct *work

work struct

void **unregister_all_irq_handlers**(struct amdgpu_device *adev)

Cleans up handlers from the DM IRQ table

Parameters

struct amdgpu_device *adev

The base driver device containing the DM device

Description

Go through low and high context IRQ tables and deallocate handlers.

void ***amdgpu_dm_irq_register_interrupt**(struct amdgpu_device *adev, struct dc_interrupt_params *int_params, void (*ih)(void*), void *handler_args)

Register a handler within DM.

Parameters

struct amdgpu_device *adev

The base driver device containing the DM device.

struct dc_interrupt_params *int_params

Interrupt parameters containing the source, and handler context

void (*ih)(void *)

Function pointer to the interrupt handler to register

void *handler_args

Arguments passed to the handler when the interrupt occurs

Description

Register an interrupt handler for the given IRQ source, under the given context. The context can either be high or low. High context handlers are executed directly within ISR context, while low context is executed within a workqueue, thereby allowing operations that sleep.

Registered handlers are called in a FIFO manner, i.e. the most recently registered handler will be called first.

Return

Handler data *struct amdgpu_dm_irq_handler_data* containing the IRQ

source, handler function, and args

void **amdgpu_dm_irq_unregister_interrupt**(struct amdgpu_device *adev, enum dc_irq_source irq_source, void *ih)

Remove a handler from the DM IRQ table

Parameters

struct amdgpu_device *adev

The base driver device containing the DM device

enum dc_irq_source irq_source

IRQ source to remove the given handler from

void *ih

Function pointer to the interrupt handler to unregister

Description

Go through both low and high context IRQ tables, and find the given handler for the given irq source. If found, remove it. Otherwise, do nothing.

int **amdgpu_dm_irq_init**(struct amdgpu_device *adev)

Initialize DM IRQ management

Parameters

struct amdgpu_device *adev

The base driver device containing the DM device

Description

Initialize DM's high and low context IRQ tables.

The N by M table contains N IRQ sources, with M *struct amdgpu_dm_irq_handler_data* hooked together in a linked list. The list_heads are initialized here. When an interrupt n is triggered, all m handlers are called in sequence, FIFO according to registration order.

The low context table requires special steps to initialize, since handlers will be deferred to a workqueue. See struct *irq_list_head*.

void **amdgpu_dm_irq_fini**(struct amdgpu_device *adev)

Tear down DM IRQ management

Parameters

struct amdgpu_device *adev

The base driver device containing the DM device

Description

Flush all work within the low context IRQ table.

int **amdgpu_dm_irq_handler**(struct amdgpu_device *adev, struct amdgpu_irq_src *source,
struct amdgpu_iv_entry *entry)

Generic DM IRQ handler

Parameters

struct amdgpu_device *adev

amdgpu base driver device containing the DM device

struct amdgpu_irq_src *source

Unused

struct amdgpu_iv_entry *entry

Data about the triggered interrupt

Description

Calls all registered high irq work immediately, and schedules work for low irq. The DM IRQ table is used to find the corresponding handlers.

void **amdgpu_dm_hpd_init**(struct amdgpu_device *adev)
hpd setup callback.

Parameters

struct amdgpu_device *adev

amdgpu_device pointer

Description

Setup the hpd pins used by the card (evergreen+). Enable the pin, set the polarity, and enable the hpd interrupts.

void **amdgpu_dm_hpd_fini**(struct amdgpu_device *adev)
hpd tear down callback.

Parameters

struct amdgpu_device *adev

amdgpu_device pointer

Description

Tear down the hpd pins used by the card (evergreen+). Disable the hpd interrupts.

void **dm_pflip_high_irq**(void *interrupt_params)
Handle pageflip interrupt

Parameters

void *interrupt_params

ignored

Description

Handles the pageflip interrupt by notifying all interested parties that the pageflip has been completed.

void dm_crtc_high_irq(void *interrupt_params)

Handles CRTC interrupt

Parameters

void *interrupt_params

used for determining the CRTC instance

Description

Handles the CRTC/VSYNC interrupt by notifying DRM's VBLANK event handler.

Atomic Implementation

WIP

void amdgpu_dm_atomic_commit_tail(struct *drm_atomic_state* *state)

AMDgpu DM's commit tail implementation.

Parameters

struct drm_atomic_state *state

The atomic state to commit

Description

This will tell DC to commit the constructed DC state from `atomic_check`, programming the hardware. Any failures here implies a hardware failure, since atomic check should have filtered anything non-kosher.

int amdgpu_dm_atomic_check(struct *drm_device* *dev, struct *drm_atomic_state* *state)

Atomic check implementation for AMDgpu DM.

Parameters

struct drm_device *dev

The DRM device

struct drm_atomic_state *state

The atomic state to commit

Description

Validate that the given atomic state is programmable by DC into hardware. This involves constructing a `struct dc_state` reflecting the new hardware state we wish to commit, then querying DC to see if it is programmable. It's important not to modify the existing DC state. Otherwise, `atomic_check` may unexpectedly commit hardware changes.

When validating the DC state, it's important that the right locks are acquired. For full updates case which removes/adds/updates streams on one CRTC while flipping on another CRTC, acquiring global lock will guarantee that any such full update commit will wait for completion of any outstanding flip using DRMs synchronization events.

Note that DM adds the affected connectors for all CRTC's in state, when that might not seem necessary. This is because DC stream creation requires the DC sink, which is tied to the DRM connector state. Cleaning this up should be possible but non-trivial - a possible TODO item.

Return

-Error code if validation failed.

Color Management Properties

The DC interface to HW gives us the following color management blocks per pipe (surface):

- Input gamma LUT (de-normalized)
- Input CSC (normalized)
- Surface degamma LUT (normalized)
- Surface CSC (normalized)
- Surface regamma LUT (normalized)
- Output CSC (normalized)

But these aren't a direct mapping to DRM color properties. The current DRM interface exposes CRTC degamma, CRTC CTM and CRTC regamma while our hardware is essentially giving:

Plane CTM -> Plane degamma -> Plane CTM -> Plane regamma -> Plane CTM

The input gamma LUT block isn't really applicable here since it operates on the actual input data itself rather than the HW fp representation. The input and output CSC blocks are technically available to use as part of the DC interface but are typically used internally by DC for conversions between color spaces. These could be blended together with user adjustments in the future but for now these should remain untouched.

The pipe blending also happens after these blocks so we don't actually support any CRTC props with correct blending with multiple planes - but we can still support CRTC color management properties in DM in most single plane cases correctly with clever management of the DC interface in DM.

As per DRM documentation, blocks should be in hardware bypass when their respective property is set to NULL. A linear DGM/RGM LUT should also be considered as putting the respective block into bypass mode.

This means that the following configuration is assumed to be the default:

Plane DGM Bypass -> Plane CTM Bypass -> Plane RGM Bypass -> ... CRTC DGM Bypass -> CRTC CTM Bypass -> CRTC RGM Bypass

void **amdgpu_dm_init_color_mod**(void)

Initialize the color module.

Parameters

void

no arguments

Description

We're not using the full color module, only certain components. Only call setup functions for components that we need.

```
const struct drm_color_lut *__extract_blob_lut(const struct drm_property_blob *blob,
                                              uint32_t *size)
```

Extracts the DRM lut and lut size from a blob.

Parameters

const struct drm_property_blob *blob
DRM color mgmt property blob

uint32_t *size
lut size

Return

DRM LUT or NULL

```
bool __is_lut_linear(const struct drm_color_lut *lut, uint32_t size)
    check if the given lut is a linear mapping of values
```

Parameters

const struct drm_color_lut *lut
given lut to check values

uint32_t size
lut size

Description

It is considered linear if the lut represents: $f(a) = (0xFF00/MAX_COLOR_LUT_ENTRIES-1)a$; for integer a in $[0, MAX_COLOR_LUT_ENTRIES)$

Return

True if the given lut is a linear mapping of values, i.e. it acts like a bypass LUT. Otherwise, false.

```
void __drm_lut_to_dc_gamma(const struct drm_color_lut *lut, struct dc_gamma *gamma, bool
                          is_legacy)
    convert the drm_color_lut to dc_gamma.
```

Parameters

const struct drm_color_lut *lut
DRM lookup table for color conversion

struct dc_gamma *gamma
DC gamma to set entries

bool is_legacy
legacy or atomic gamma

Description

The conversion depends on the size of the lut - whether or not it's legacy.

```
void __drm_ctm_to_dc_matrix(const struct drm_color_ctm *ctm, struct fixed31_32 *matrix)
    converts a DRM CTM to a DC CSC float matrix
```

Parameters

const struct drm_color_ctm *ctm
DRM color transformation matrix

struct fixed31_32 *matrix
DC CSC float matrix

Description

The matrix needs to be a 3x4 (12 entry) matrix.

int __set_legacy_tf(struct dc_transfer_func *func, const struct drm_color_lut *lut, uint32_t lut_size, bool has_rom)

Calculates the legacy transfer function

Parameters

struct dc_transfer_func *func
transfer function

const struct drm_color_lut *lut
lookup table that defines the color space

uint32_t lut_size
size of respective lut

bool has_rom
if ROM can be used for hardcoded curve

Description

Only for sRGB input space

Return

0 in case of success, -ENOMEM if fails

int __set_output_tf(struct dc_transfer_func *func, const struct drm_color_lut *lut, uint32_t lut_size, bool has_rom)

calculates the output transfer function based on expected input space.

Parameters

struct dc_transfer_func *func
transfer function

const struct drm_color_lut *lut
lookup table that defines the color space

uint32_t lut_size
size of respective lut

bool has_rom
if ROM can be used for hardcoded curve

Return

0 in case of success. -ENOMEM if fails.

int __set_input_tf(struct dc_transfer_func *func, const struct drm_color_lut *lut, uint32_t lut_size)

calculates the input transfer function based on expected input space.

Parameters

struct dc_transfer_func *func
transfer function

const struct drm_color_lut *lut
lookup table that defines the color space

uint32_t lut_size
size of respective lut.

Return

0 in case of success. -ENOMEM if fails.

int **amdgpu_dm_verify_lut_sizes**(const struct [drm_crtc_state](#) *crtc_state)
verifies if DRM luts match the hw supported sizes

Parameters

const struct drm_crtc_state *crtc_state
the DRM CRTC state

Description

Verifies that the Degamma and Gamma LUTs attached to the `crtc_state` are of the expected size.

Return

0 on success. -EINVAL if any lut sizes are invalid.

int **amdgpu_dm_update_crtc_color_mgmt**(struct `dm_crtc_state` *crtc)
Maps DRM color management to DC stream.

Parameters

struct dm_crtc_state *crtc
amdgpu_dm crtc state

Description

With no plane level color management properties we're free to use any of the HW blocks as long as the CRTC CTM always comes before the CRTC RGM and after the CRTC DGM.

- The CRTC RGM block will be placed in the RGM LUT block if it is non-linear.
- The CRTC DGM block will be placed in the DGM LUT block if it is non-linear.
- The CRTC CTM will be placed in the gamut remap block if it is non-linear.

The RGM block is typically more fully featured and accurate across all ASICs - DCE can't support a custom non-linear CRTC DGM.

For supporting both plane level color management and CRTC level color management at once we have to either restrict the usage of CRTC properties or blend adjustments together.

Return

0 on success. Error code if setup fails.

```
int amdgpu_dm_update_plane_color_mgmt(struct dm_crtc_state *crtc, struct dc_plane_state
                                     *dc_plane_state)
```

Maps DRM color management to DC plane.

Parameters

```
struct dm_crtc_state *crtc
    amdgpu_dm crtc state
```

```
struct dc_plane_state *dc_plane_state
    target DC surface
```

Description

Update the underlying `dc_stream_state`'s input transfer function (ITF) in preparation for hardware commit. The transfer function used depends on the preparation done on the stream for color management.

Return

0 on success. -ENOMEM if mem allocation fails.

DC Color Capabilities between DCN generations

DRM/KMS framework defines three CRTC color correction properties: degamma, color transformation matrix (CTM) and gamma, and two properties for degamma and gamma LUT sizes. AMD DC programs some of the color correction features pre-blending but DRM/KMS has not per-plane color correction properties.

In general, the DRM CRTC color properties are programmed to DC, as follows: CRTC gamma after blending, and CRTC degamma pre-blending. Although CTM is programmed after blending, it is mapped to DPP hw blocks (pre-blending). Other color caps available in the hw is not currently exposed by DRM interface and are bypassed.

Color management caps (DPP and MPC)

Modules/color calculates various color operations which are translated to abstracted HW. DCE 5-12 had almost no important changes, but starting with DCN1, every new generation comes with fairly major differences in color pipeline. Therefore, we abstract color pipe capabilities so modules/DM can decide mapping to HW block based on logical capabilities.

```
struct rom_curve_caps
    predefined transfer function caps for degamma and regamma
```

Definition:

```
struct rom_curve_caps {
    uint16_t srgb : 1;
    uint16_t bt2020 : 1;
    uint16_t gamma2_2 : 1;
    uint16_t pq : 1;
    uint16_t hlg : 1;
};
```

Members

srgb

RGB color space transfer func

bt2020

BT.2020 transfer func

gamma2_2

standard gamma

pq

perceptual quantizer transfer function

hlg

hybrid log-gamma transfer function

struct **dpp_color_caps**

color pipeline capabilities for display pipe and plane blocks

Definition:

```
struct dpp_color_caps {
    uint16_t dcn_arch : 1;
    uint16_t input_lut_shared : 1;
    uint16_t icsc : 1;
    uint16_t dgam_ram : 1;
    uint16_t post_csc : 1;
    uint16_t gamma_corr : 1;
    uint16_t hw_3d_lut : 1;
    uint16_t ogam_ram : 1;
    uint16_t ocsc : 1;
    uint16_t dgam_rom_for_yuv : 1;
    struct rom_curve_caps dgam_rom_caps;
    struct rom_curve_caps ogam_rom_caps;
};
```

Members**dcn_arch**

all DCE generations treated the same

input_lut_shared

shared with DGAM. Input LUT is different than most LUTs, just plain 256-entry lookup

icsc

input color space conversion

dgam_ram

programmable degamma LUT

post_csc

post color space conversion, before gamut remap

gamma_corr

degamma correction

hw_3d_lut

3D LUT support. It implies a shaper LUT before. It may be shared with MPC by setting mpc:shared_3d_lut flag

ogam_ram

programmable out/blend gamma LUT

ocsc

output color space conversion

dgam_rom_for_yuv

pre-defined degamma LUT for YUV planes

dgam_rom_caps

pre-defined curve caps for degamma 1D LUT

ogam_rom_caps

pre-defined curve caps for regamma 1D LUT

Note

hdr_mult and gamut remap (CTM) are always available in DPP (in that order)

struct **mpc_color_caps**

color pipeline capabilities for multiple pipe and plane combined blocks

Definition:

```
struct mpc_color_caps {
    uint16_t gamut_remap : 1;
    uint16_t ogam_ram : 1;
    uint16_t ocsc : 1;
    uint16_t num_3dluts : 3;
    uint16_t shared_3d_lut:1;
    struct rom_curve_caps ogam_rom_caps;
};
```

Members

gamut_remap

color transformation matrix

ogam_ram

programmable out gamma LUT

ocsc

output color space conversion matrix

num_3dluts

MPC 3D LUT; always assumes a preceding shaper LUT

shared_3d_lut

shared 3D LUT flag. Can be either DPP or MPC, but single instance

ogam_rom_caps

pre-defined curve caps for regamma 1D LUT

struct **dc_color_caps**

color pipes capabilities for DPP and MPC hw blocks

Definition:

```
struct dc_color_caps {
    struct dpp_color_caps dpp;
    struct mpc_color_caps mpc;
};
```

Members

dpp

color pipes caps for DPP

mpc

color pipes caps for MPC

enum **pipe_split_policy**

Pipe split strategy supported by DCN

Constants

MPC_SPLIT_DYNAMIC

DC will automatically decide how to split the pipe in order to bring the best trade-off between performance and power consumption. This is the recommended option.

MPC_SPLIT_AVOID

Avoid pipe split, which means that DC will not try any sort of split optimization.

MPC_SPLIT_AVOID_MULT_DISP

With this option, DC will only try to optimize the pipe utilization when using a single display; if the user connects to a second display, DC will avoid pipe split.

Description

This enum is used to define the pipe split policy supported by DCN. By default, DC favors MPC_SPLIT_DYNAMIC.

struct **dc_validation_set**

Struct to store surface/stream associations for validation

Definition:

```
struct dc_validation_set {
    struct dc_stream_state *stream;
    struct dc_plane_state *plane_states[MAX_SURFACES];
    uint8_t plane_count;
};
```

Members

stream

Stream state properties

plane_states

Surface state

plane_count

Total of active planes

The color pipeline has undergone major changes between DCN hardware generations. What's possible to do before and after blending depends on hardware capabilities, as illustrated below by the DCN 2.0 and DCN 3.0 families schemas.

DCN 2.0 family color caps and mapping

DCN 3.0 family color caps and mapping

Blend Mode Properties

Pixel blend mode is a DRM plane composition property of *drm_plane* used to describes how pixels from a foreground plane (fg) are composited with the background plane (bg). Here, we present main concepts of DRM blend mode to help to understand how this property is mapped to AMD DC interface. See more about this DRM property and the alpha blending equations in *DRM Plane Composition Properties*.

Basically, a blend mode sets the alpha blending equation for plane composition that fits the mode in which the alpha channel affects the state of pixel color values and, therefore, the resulted pixel color. For example, consider the following elements of the alpha blending equation:

- *fg.rgb*: Each of the RGB component values from the foreground's pixel.
- *fg.alpha*: Alpha component value from the foreground's pixel.
- *bg.rgb*: Each of the RGB component values from the background.
- *plane_alpha*: Plane alpha value set by the **plane "alpha" property**, see more in *DRM Plane Composition Properties*.

in the basic alpha blending equation:

$$\text{out.rgb} = \text{alpha} * \text{fg.rgb} + (1 - \text{alpha}) * \text{bg.rgb}$$

the alpha channel value of each pixel in a plane is ignored and only the plane alpha affects the resulted pixel color values.

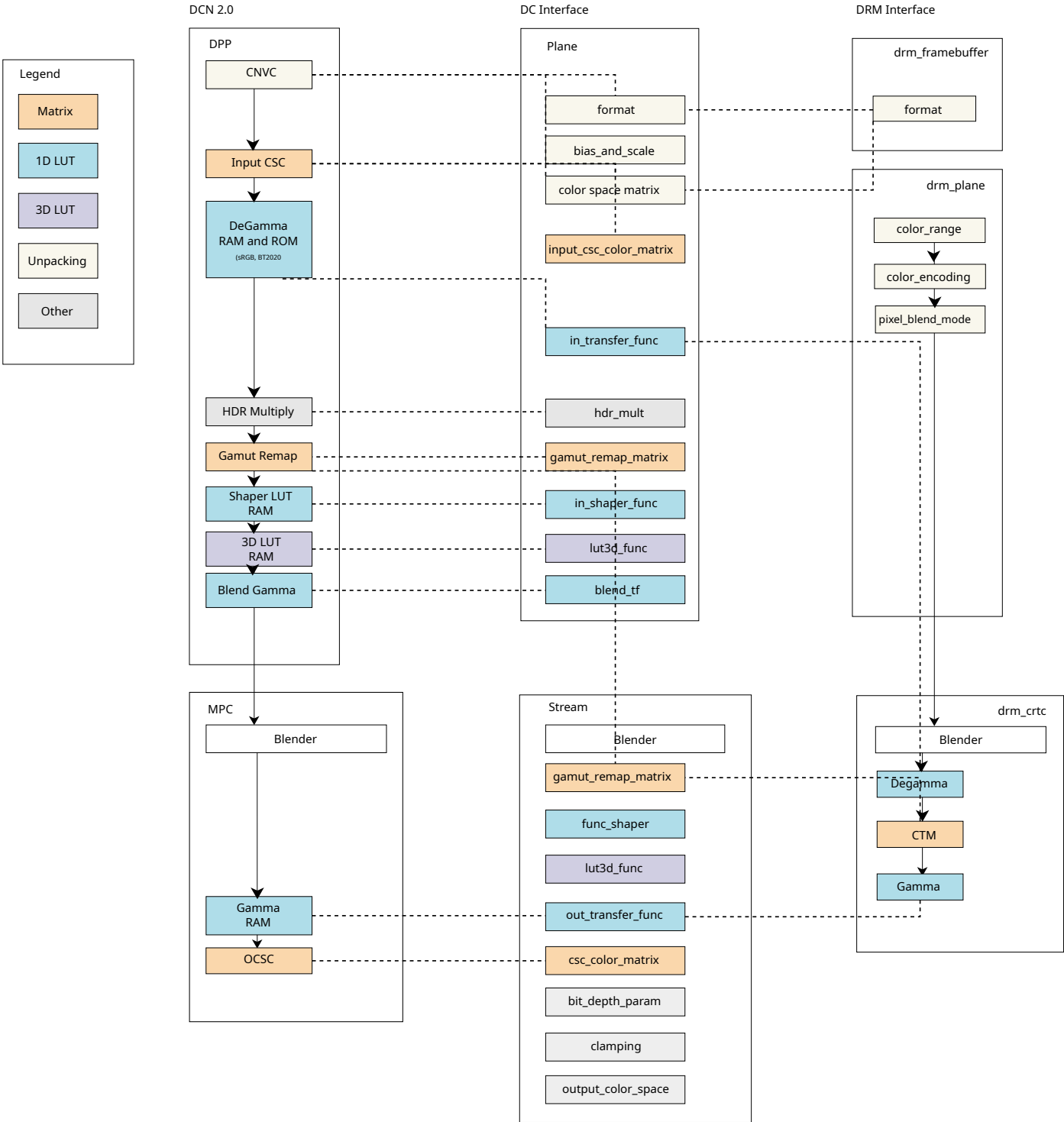
DRM has three blend mode to define the blend formula in the plane composition:

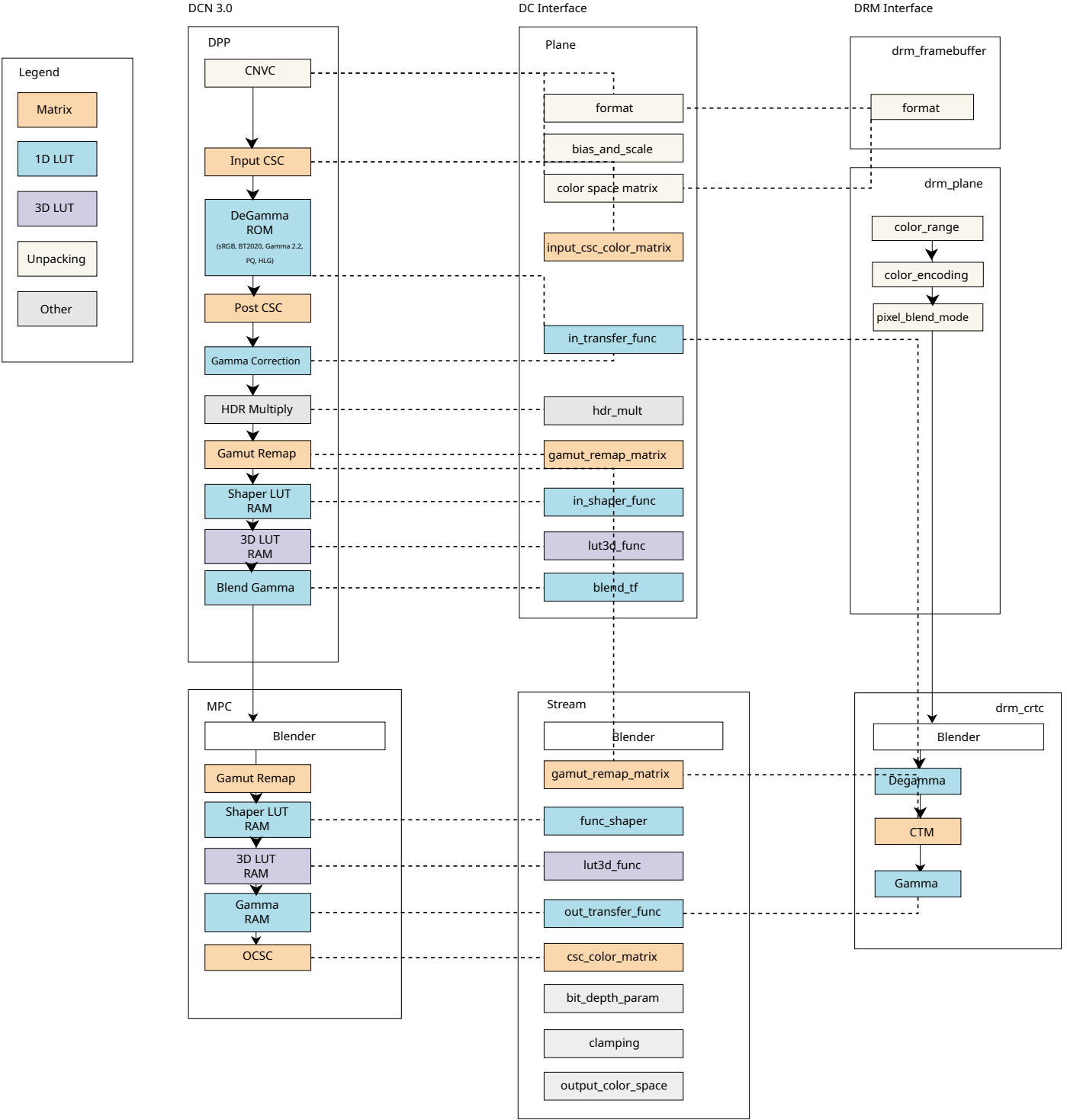
- **None**: Blend formula that ignores the pixel alpha.
- **Pre-multiplied**: Blend formula that assumes the pixel color values in a plane was already pre-multiplied by its own alpha channel before storage.
- **Coverage**: Blend formula that assumes the pixel color values were not pre-multiplied with the alpha channel values.

and pre-multiplied is the default pixel blend mode, that means, when no blend mode property is created or defined, DRM considers the plane's pixels has pre-multiplied color values. On IGT GPU tools, the *kms_plane_alpha_blend* test provides a set of subtests to verify plane alpha and blend mode properties.

The DRM blend mode and its elements are then mapped by AMDGPU display manager (DM) to program the blending configuration of the Multiple Pipe/Plane Combined (MPC), as follows:

Multiple Pipe/Plane Combined (MPC) is a component in the hardware pipeline that performs blending of multiple planes, using global and per-pixel alpha. It also performs post-blending color correction operations according to the hardware capabilities, such as color transformation matrix and gamma 1D and 3D LUT.





struct mpcc_blnd_cfg

MPCC blending configuration

Definition:

```
struct mpcc_blnd_cfg {
    struct tg_color black_color;
    enum mpcc_alpha_blend_mode alpha_mode;
    bool pre_multiplied_alpha;
    int global_gain;
    int global_alpha;
    bool overlap_only;
    int bottom_gain_mode;
    int background_color_bpc;
    int top_gain;
    int bottom_inside_gain;
    int bottom_outside_gain;
};
```

Members**black_color**

background color

alpha_mode

alpha blend mode (MPCC_ALPHA_BLND_MODE)

pre_multiplied_alpha

whether pixel color values were pre-multiplied by the alpha channel (MPCC_ALPHA_MULTIPLIED_MODE)

global_gain

used when blend mode considers both pixel alpha and plane alpha value and assumes the global alpha value.

global_alpha

plane alpha value

Therefore, the blending configuration for a single MPCC instance on the MPC tree is defined by *mpcc_blnd_cfg*, where *pre_multiplied_alpha* is the alpha pre-multiplied mode flag used to set MPCC_ALPHA_MULTIPLIED_MODE. It controls whether alpha is multiplied (true/false), being only true for DRM pre-multiplied blend mode. *mpcc_alpha_blend_mode* defines the alpha blend mode regarding pixel alpha and plane alpha values. It sets one of the three modes for MPCC_ALPHA_BLND_MODE, as described below.

enum mpcc_alpha_blend_mode

define the alpha blend mode regarding pixel alpha and plane alpha values

Constants**MPCC_ALPHA_BLEND_MODE_PER_PIXEL_ALPHA**

per pixel alpha using DPP alpha value

MPCC_ALPHA_BLEND_MODE_PER_PIXEL_ALPHA_COMBINED_GLOBAL_GAIN

per pixel alpha using DPP alpha value multiplied by a global gain (plane alpha)

MPCC_ALPHA_BLEND_MODE_GLOBAL_ALPHA

global alpha value, ignores pixel alpha and consider only plane alpha

DM then maps the elements of `enum mpcc_alpha_blend_mode` to those in the DRM blend formula, as follows:

- *MPC pixel alpha* matches *DRM fg.alpha* as the alpha component value from the plane's pixel
- *MPC global alpha* matches *DRM plane_alpha* when the pixel alpha should be ignored and, therefore, pixel values are not pre-multiplied
- *MPC global gain* assumes *MPC global alpha* value when both *DRM fg.alpha* and *DRM plane_alpha* participate in the blend equation

In short, *fg.alpha* is ignored by selecting `MPCC_ALPHA_BLEND_MODE_GLOBAL_ALPHA`. On the other hand, $(\text{plane_alpha} * \text{fg.alpha})$ component becomes available by selecting `MPCC_ALPHA_BLEND_MODE_PER_PIXEL_ALPHA_COMBINED_GLOBAL_GAIN`. And the `MPCC_ALPHA_MULTIPLIED_MODE` defines if the pixel color values are pre-multiplied by alpha or not.

Blend configuration flow

The alpha blending equation is configured from DRM to DC interface by the following path:

1. When updating a `drm_plane_state`, DM calls `amdgpu_dm_plane_fill_blending_from_plane_state` that maps `drm_plane_state` attributes to `dc_plane_info` struct to be handled in the OS-agnostic component (DC).
2. On DC interface, `struct mpcc_blnd_cfg` programs the MPCC blend configuration considering the `dc_plane_info` input from DPP.

Display Core Debug tools

DC Visual Confirmation

Display core provides a feature named visual confirmation, which is a set of bars added at the scanout time by the driver to convey some specific information. In general, you can enable this debug option by using:

```
echo <N> > /sys/kernel/debug/dri/0/amdgpu_dm_visual_confirm
```

Where *N* is an integer number for some specific scenarios that the developer wants to enable, you will see some of these debug cases in the following subsection.

Multiple Planes Debug

If you want to enable or debug multiple planes in a specific user-space application, you can leverage a debug feature named visual confirm. For enabling it, you will need:

```
echo 1 > /sys/kernel/debug/dri/0/amdgpu_dm_visual_confirm
```

You need to reload your GUI to see the visual confirmation. When the plane configuration changes or a full update occurs there will be a colored bar at the bottom of each hardware plane being drawn on the screen.

- The color indicates the format - For example, red is AR24 and green is NV12
- The height of the bar indicates the index of the plane
- Pipe split can be observed if there are two bars with a difference in height covering the same plane

Consider the video playback case in which a video is played in a specific plane, and the desktop is drawn in another plane. The video plane should feature one or two green bars at the bottom of the video depending on pipe split configuration.

- There should **not** be any visual corruption
- There should **not** be any underflow or screen flashes
- There should **not** be any black screens
- There should **not** be any cursor corruption
- Multiple plane **may** be briefly disabled during window transitions or resizing but should come back after the action has finished

Pipe Split Debug

Sometimes we need to debug if DCN is splitting pipes correctly, and visual confirmation is also handy for this case. Similar to the MPO case, you can use the below command to enable visual confirmation:

```
echo 1 > /sys/kernel/debug/dri/0/amdgpu_dm_visual_confirm
```

In this case, if you have a pipe split, you will see one small red bar at the bottom of the display covering the entire display width and another bar covering the second pipe. In other words, you will see a bit high bar in the second pipe.

DTN Debug

DC (DCN) provides an extensive log that dumps multiple details from our hardware configuration. Via debugfs, you can capture those status values by using Display Test Next (DTN) log, which can be captured via debugfs by using:

```
cat /sys/kernel/debug/dri/0/amdgpu_dm_dtn_log
```

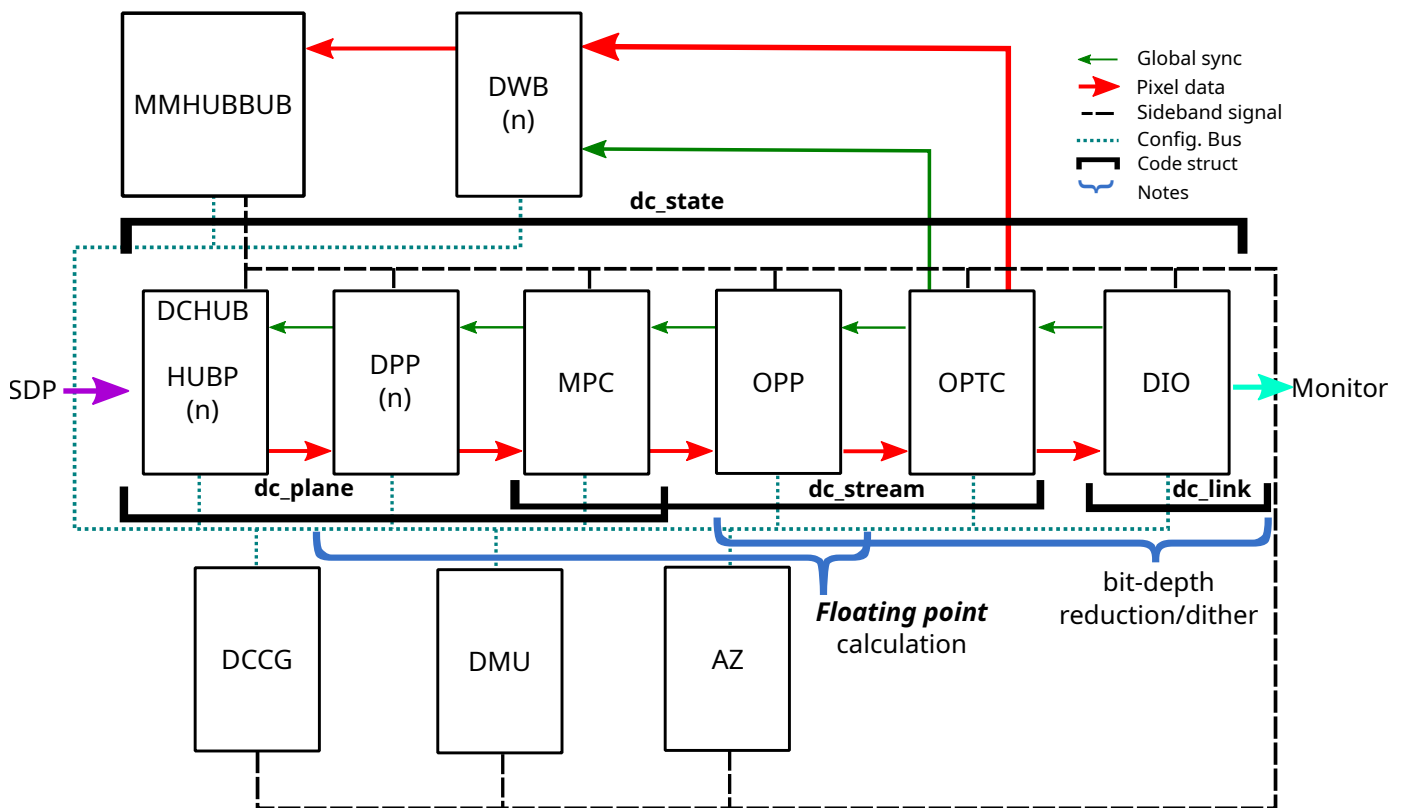
Since this log is updated accordingly with DCN status, you can also follow the change in real-time by using something like:

```
sudo watch -d cat /sys/kernel/debug/dri/0/amdgpu_dm_dtn_log
```

When reporting a bug related to DC, consider attaching this log before and after you reproduce the bug.

Display Core Next (DCN)

To equip our readers with the basic knowledge of how AMD Display Core Next (DCN) works, we need to start with an overview of the hardware pipeline. Below you can see a picture that provides a DCN overview, keep in mind that this is a generic diagram, and we have variations per ASIC.



Based on this diagram, we can pass through each block and briefly describe them:

- **Display Controller Hub (DCHUB):** This is the gateway between the Scalable Data Port (SDP) and DCN. This component has multiple features, such as memory arbitration, rotation, and cursor manipulation.
- **Display Pipe and Plane (DPP):** This block provides pre-blend pixel processing such as color space conversion, linearization of pixel data, tone mapping, and gamut mapping.
- **Multiple Pipe/Plane Combined (MPC):** This component performs blending of multiple planes, using global or per-pixel alpha.
- **Output Pixel Processing (OPP):** Process and format pixels to be sent to the display.
- **Output Pipe Timing Combiner (OPTC):** It generates time output to combine streams or divide capabilities. CRC values are generated in this block.

- **Display Output (DIO):** Codify the output to the display connected to our GPU.
- **Display Writeback (DWB):** It provides the ability to write the output of the display pipe back to memory as video frames.
- **Multi-Media HUB (MMHUBBUB):** Memory controller interface for DMCUB and DWB (Note that DWB is not hooked yet).
- **DCN Management Unit (DMU):** It provides registers with access control and interrupts the controller to the SOC host interrupt unit. This block includes the Display Micro-Controller Unit - version B (DMCUB), which is handled via firmware.
- **DCN Clock Generator Block (DCCG):** It provides the clocks and resets for all of the display controller clock domains.
- **Azalia (AZ):** Audio engine.

The above diagram is an architecture generalization of DCN, which means that every ASIC has variations around this base model. Notice that the display pipeline is connected to the Scalable Data Port (SDP) via DCHUB; you can see the SDP as the element from our Data Fabric that feeds the display pipe.

Always approach the DCN architecture as something flexible that can be configured and reconfigured in multiple ways; in other words, each block can be setup or ignored accordingly with userspace demands. For example, if we want to drive an [8k@60Hz](#) with a DSC enabled, our DCN may require 4 DPP and 2 OPP. It is DC's responsibility to drive the best configuration for each specific scenario. Orchestrate all of these components together requires a sophisticated communication interface which is highlighted in the diagram by the edges that connect each block; from the chart, each connection between these blocks represents:

1. Pixel data interface (red): Represents the pixel data flow;
2. Global sync signals (green): It is a set of synchronization signals composed by VStartup, VUpdate, and VReady;
3. Config interface: Responsible to configure blocks;
4. Sideband signals: All other signals that do not fit the previous one.

These signals are essential and play an important role in DCN. Nevertheless, the Global Sync deserves an extra level of detail described in the next section.

All of these components are represented by a data structure named `dc_state`. From DCHUB to MPC, we have a representation called `dc_plane`; from MPC to OPTC, we have `dc_stream`, and the output (DIO) is handled by `dc_link`. Keep in mind that HUBP accesses a surface using a specific format read from memory, and our `dc_plane` should work to convert all pixels in the plane to something that can be sent to the display via `dc_stream` and `dc_link`.

Front End and Back End

Display pipeline can be broken down into two components that are usually referred as **Front End (FE)** and **Back End (BE)**, where FE consists of:

- DCHUB (Mainly referring to a subcomponent named HUBP)
- DPP
- MPC

On the other hand, BE consist of

- OPP
- OPTC
- DIO (DP/HDMI stream encoder and link encoder)

OPP and OPTC are two joining blocks between FE and BE. On a side note, this is a one-to-one mapping of the link encoder to PHY, but we can configure the DCN to choose which link encoder to connect to which PHY. FE's main responsibility is to change, blend and compose pixel data, while BE's job is to frame a generic pixel stream to a specific display's pixel stream.

Data Flow

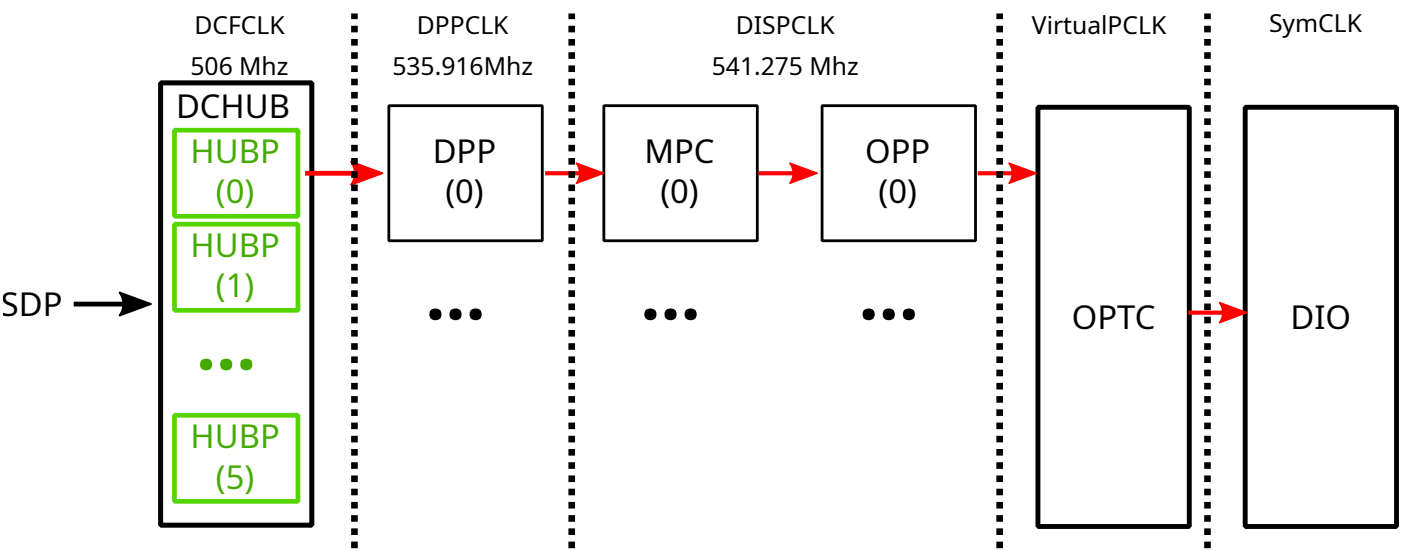
Initially, data is passed in from VRAM through Data Fabric (DF) in native pixel formats. Such data format stays through till HUBP in DCHUB, where HUBP unpacks different pixel formats and outputs them to DPP in uniform streams through 4 channels (1 for alpha + 3 for colors).

The Converter and Cursor (CNVC) in DPP would then normalize the data representation and convert them to a DCN specific floating-point format (i.e., different from the IEEE floating-point format). In the process, CNVC also applies a degamma function to transform the data from non-linear to linear space to relax the floating-point calculations following. Data would stay in this floating-point format from DPP to OPP.

Starting OPP, because color transformation and blending have been completed (i.e alpha can be dropped), and the end sinks do not require the precision and dynamic range that floating points provide (i.e. all displays are in integer depth format), bit-depth reduction/dithering would kick in. In OPP, we would also apply a regamma function to introduce the gamma removed earlier back. Eventually, we output data in integer format at DIO.

AMD Hardware Pipeline

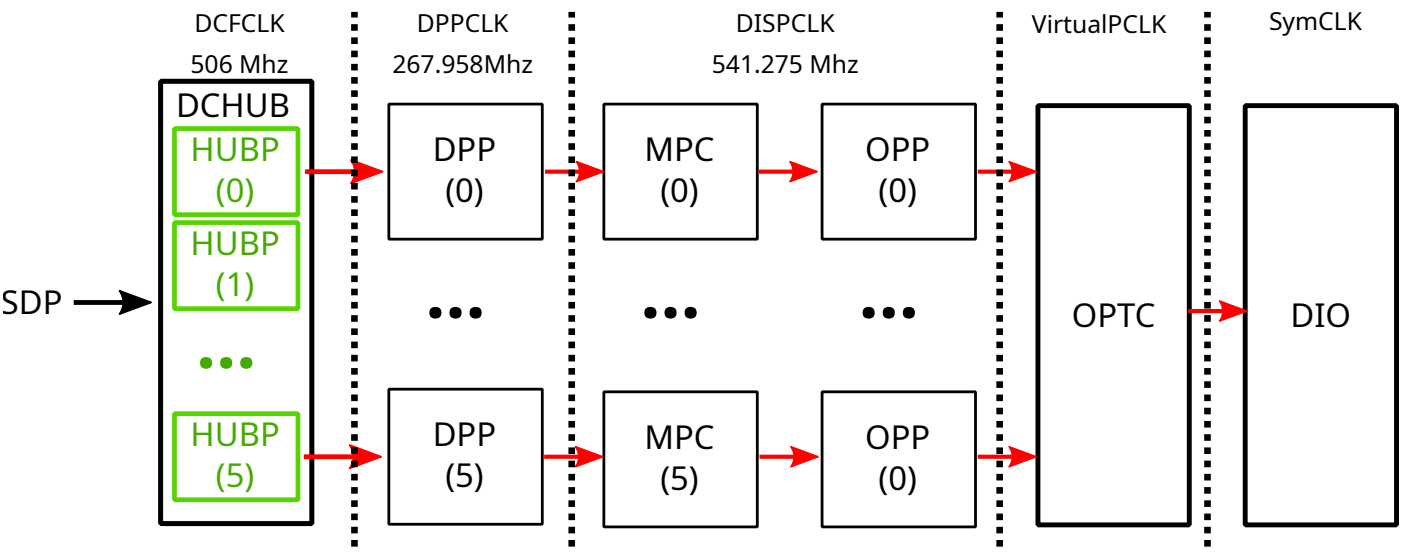
When discussing graphics on Linux, the **pipeline** term can sometimes be overloaded with multiple meanings, so it is important to define what we mean when we say **pipeline**. In the DCN driver, we use the term **hardware pipeline** or **pipeline** or just **pipe** as an abstraction to indicate a sequence of DCN blocks instantiated to address some specific configuration. DC core treats DCN blocks as individual resources, meaning we can build a pipeline by taking resources for all individual hardware blocks to compose one pipeline. In actuality, we can't connect an arbitrary block from one pipe to a block from another pipe; they are routed linearly, except for DSC, which can be arbitrarily assigned as needed. We have this pipeline concept for trying to optimize bandwidth utilization.



Additionally, let’s take a look at parts of the DTN log (see ‘[Display Core Debug tools](#)’ for more information) since this log can help us to see part of this pipeline behavior in real-time:

```
HUBP:  format  addr_hi  width  height  ...
[ 0]:    8h      81h   3840   2160
[ 1]:    0h      0h     0     0
[ 2]:    0h      0h     0     0
[ 3]:    0h      0h     0     0
[ 4]:    0h      0h     0     0
...
MPCC:  OPP   DPP  ...
[ 0]:   0h   0h  ...
```

The first thing to notice from the diagram and DTN log it is the fact that we have different clock domains for each part of the DCN blocks. In this example, we have just a single **pipeline** where the data flows from DCHUB to DIO, as we intuitively expect. Nonetheless, DCN is flexible, as mentioned before, and we can split this single pipe differently, as described in the below diagram:



Now, if we inspect the DTN log again we can see some interesting changes:

```
HUBP:  format  addr_hi  width  height ...
[ 0]:      8h      81h   1920   2160 ...
...
[ 4]:      0h      0h     0     0 ...
[ 5]:      8h      81h   1920   2160 ...
...
MPCC:  OPP    DPP ...
[ 0]:   0h    0h ...
[ 5]:   0h    5h ...
```

From the above example, we now split the display pipeline into two vertical parts of 1920x2160 (i.e., 3440x2160), and as a result, we could reduce the clock frequency in the DPP part. This is not only useful for saving power but also to better handle the required throughput. The idea to keep in mind here is that the pipe configuration can vary a lot according to the display configuration, and it is the DML's responsibility to set up all required configuration parameters for multiple scenarios supported by our hardware.

Global Sync

Many DCN registers are double buffered, most importantly the surface address. This allows us to update DCN hardware atomically for page flips, as well as for most other updates that don't require enabling or disabling of new pipes.

(Note: There are many scenarios when DC will decide to reserve extra pipes in order to support outputs that need a very high pixel clock, or for power saving purposes.)

These atomic register updates are driven by global sync signals in DCN. In order to understand how atomic updates interact with DCN hardware, and how DCN signals page flip and vblank events it is helpful to understand how global sync is programmed.

Global sync consists of three signals, VSTARTUP, VUPDATE, and VREADY. These are calculated by the Display Mode Library - DML (drivers/gpu/drm/amd/display/dc/dml) based on a large number of parameters and ensure our hardware is able to feed the DCN pipeline without underflows or hangs in any given system configuration. The global sync signals always happen during VBlank, are independent from the VSync signal, and do not overlap each other.

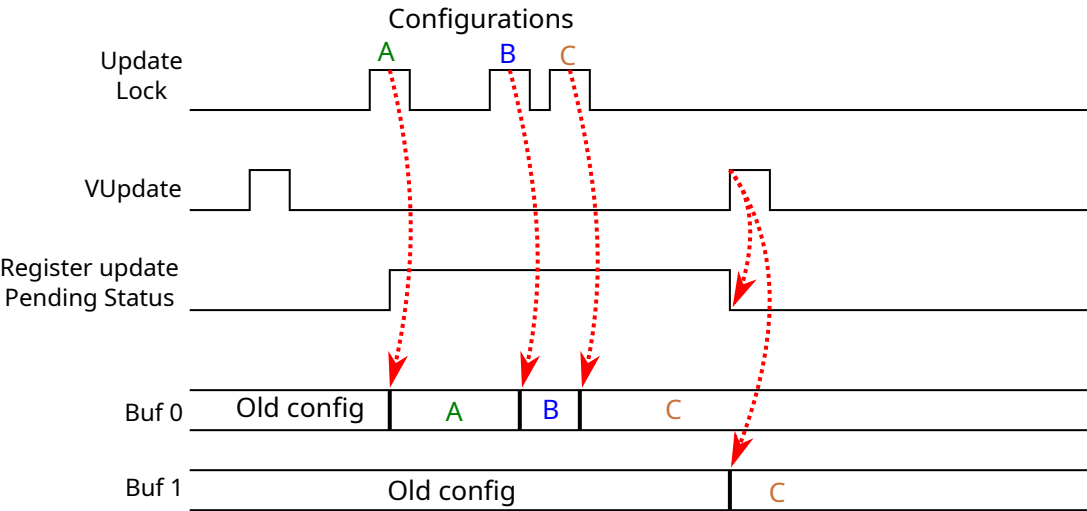
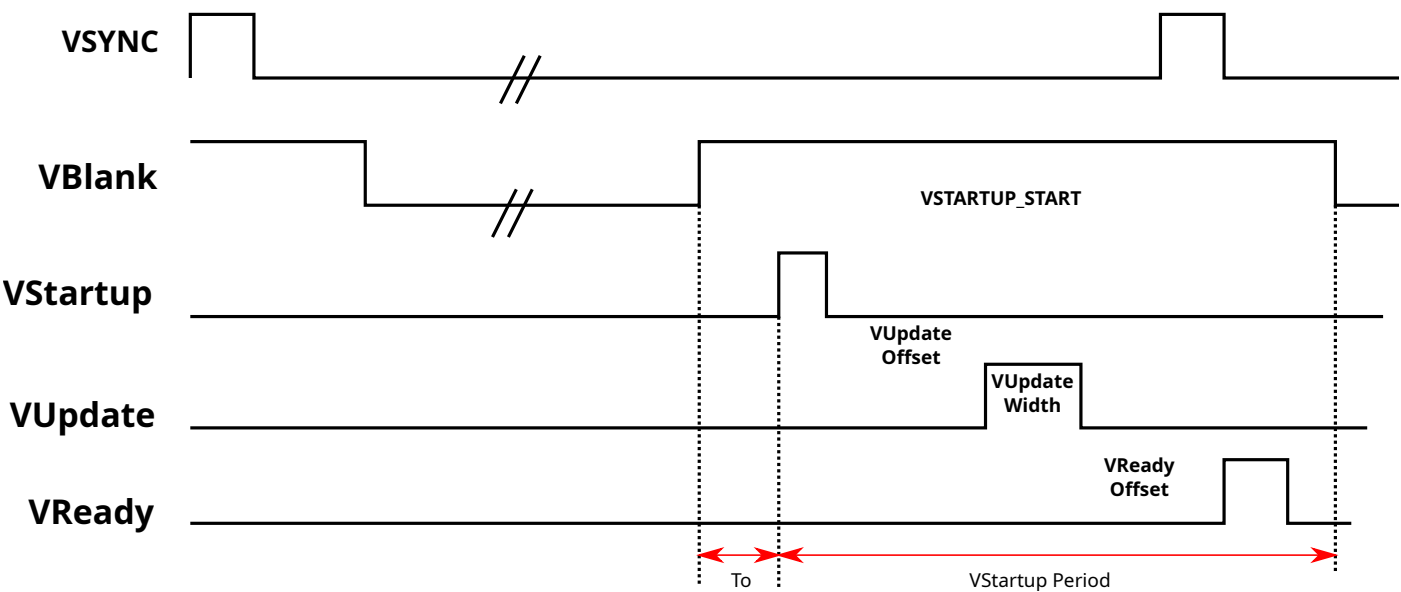
VUPDATE is the only signal that is of interest to the rest of the driver stack or userspace clients as it signals the point at which hardware latches to atomically programmed (i.e. double buffered) registers. Even though it is independent of the VSync signal we use VUPDATE to signal the VSync event as it provides the best indication of how atomic commits and hardware interact.

Since DCN hardware is double-buffered the DC driver is able to program the hardware at any point during the frame.

The below picture illustrates the global sync signals:

These signals affect core DCN behavior. Programming them incorrectly will lead to a number of negative consequences, most of them quite catastrophic.

The following picture shows how global sync allows for a mailbox style of updates, i.e. it allows for multiple re-configurations between VUpdate events where only the last configuration programmed before the VUpdate signal becomes effective.



Multiplane Overlay (MPO)

Note: You will get more from this page if you have already read the *'Display Core Next (DCN)'*.

Multiplane Overlay (MPO) allows for multiple framebuffers to be composited via fixed-function hardware in the display controller rather than using graphics or compute shaders for composition. This can yield some power savings if it means the graphics/compute pipelines can be put into low-power states. In summary, MPO can bring the following benefits:

- Decreased GPU and CPU workload - no composition shaders needed, no extra buffer copy needed, GPU can remain idle.
- Plane independent page flips - No need to be tied to global compositor page-flip present rate, reduced latency, independent timing.

Note: Keep in mind that MPO is all about power-saving; if you want to learn more about power-save in the display context, check the link: [Power](#).

Multiplane Overlay is only available using the DRM atomic model. The atomic model only uses a single userspace IOCTL for configuring the display hardware (modesetting, page-flipping, etc) - `drmModeAtomicCommit`. To query hardware resources and limitations userspace also calls into `drmModeGetResources` which reports back the number of planes, CRTC, and connectors. There are three types of DRM planes that the driver can register and work with:

- `DRM_PLANE_TYPE_PRIMARY`: Primary planes represent a “main” plane for a CRTC, primary planes are the planes operated upon by CRTC modesetting and flipping operations.
- `DRM_PLANE_TYPE_CURSOR`: Cursor planes represent a “cursor” plane for a CRTC. Cursor planes are the planes operated upon by the cursor IOCTLs
- `DRM_PLANE_TYPE_OVERLAY`: Overlay planes represent all non-primary, non-cursor planes. Some drivers refer to these types of planes as “sprites” internally.

To illustrate how it works, let's take a look at a device that exposes the following planes to userspace:

- 4 Primary planes (1 per CRTC).
- 4 Cursor planes (1 per CRTC).
- 1 Overlay plane (shared among CRTC).

Note: Keep in mind that different ASICs might expose other numbers of planes.

For this hardware example, we have 4 pipes (if you don't know what AMD pipe means, look at *'Display Core Next (DCN)'*, section “AMD Hardware Pipeline”). Typically most AMD devices operate in a pipe-split configuration for optimal single display output (e.g., 2 pipes per plane).

A typical MPO configuration from userspace - 1 primary + 1 overlay on a single display - will see 4 pipes in use, 2 per plane.

At least 1 pipe must be used per plane (primary and overlay), so for this hypothetical hardware that we are using as an example, we have an absolute limit of 4 planes across all CRTC. Atomic

commits will be rejected for display configurations using more than 4 planes. Again, it is important to stress that every DCN has different restrictions; here, we are just trying to provide the concept idea.

Plane Restrictions

AMDGPU imposes restrictions on the use of DRM planes in the driver.

Atomic commits will be rejected for commits which do not follow these restrictions:

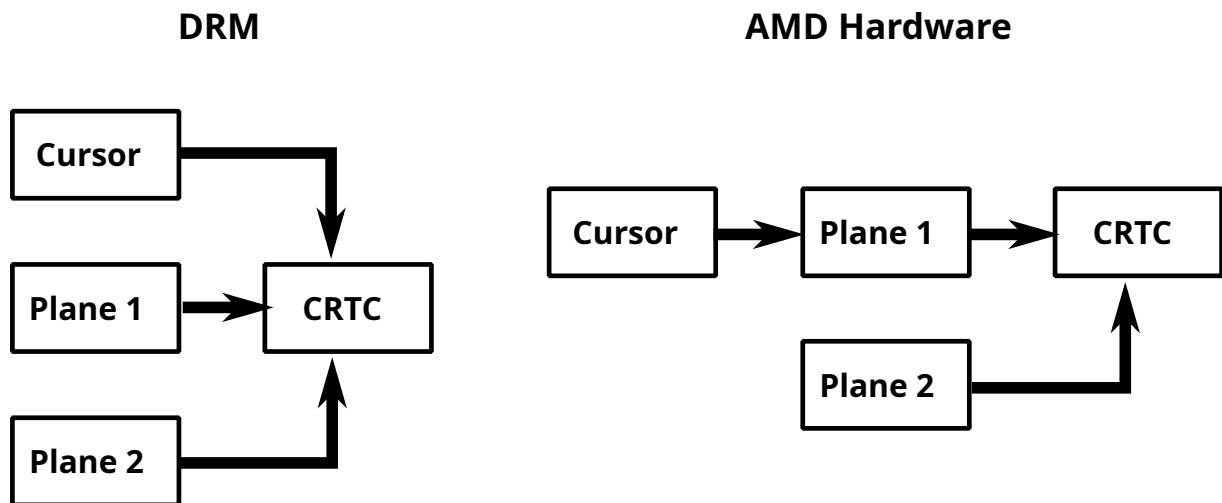
- Overlay planes must be in ARGB8888 or XRGB8888 format
- Planes cannot be placed outside of the CRTC destination rectangle
- Planes cannot be downscaled more than 1/4x of their original size
- Planes cannot be upscaled more than 16x of their original size

Not every property is available on every plane:

- Only primary planes have color-space and non-RGB format support
- Only overlay planes have alpha blending support

Cursor Restrictions

Before we start to describe some restrictions around cursor and MPO, see the below image:



The image on the left side represents how DRM expects the cursor and planes to be blended. However, AMD hardware handles cursors differently, as you can see on the right side; basically, our cursor cannot be drawn outside its associated plane as it is being treated as part of the plane. Another consequence of that is that cursors inherit the color and scale from the plane.

As a result of the above behavior, do not use legacy API to set up the cursor plane when working with MPO; otherwise, you might encounter unexpected behavior.

In short, AMD HW has no dedicated cursor planes. A cursor is attached to another plane and therefore inherits any scaling or color processing from its parent plane.

Use Cases

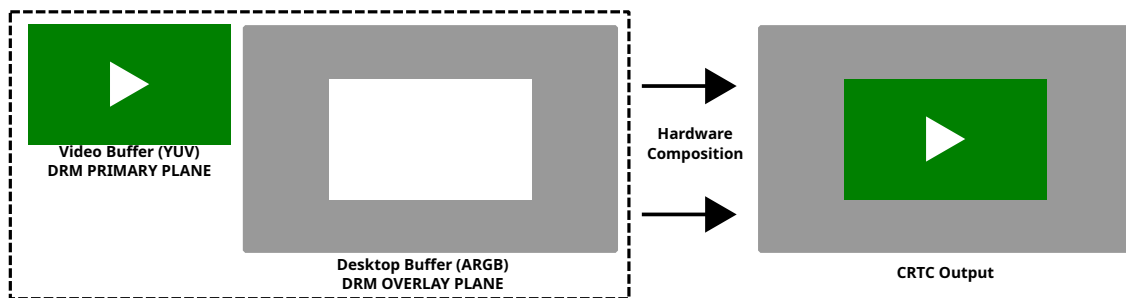
Picture-in-Picture (PIP) playback - Underlay strategy

Video playback should be done using the “primary plane as underlay” MPO strategy. This is a 2 planes configuration:

- 1 YUV DRM Primary Plane (e.g. NV12 Video)
- 1 RGBA DRM Overlay Plane (e.g. ARGB8888 desktop). The compositor should prepare the framebuffers for the planes as follows: - The overlay plane contains general desktop UI, video player controls, and video subtitles - Primary plane contains one or more videos

Note: Keep in mind that we could extend this configuration to more planes, but that is currently not supported by our driver yet (maybe if we have a userspace request in the future, we can change that).

See below a single-video example:



Note: We could extend this behavior to more planes, but that is currently not supported by our driver.

The video buffer should be used directly for the primary plane. The video can be scaled and positioned for the desktop using the properties: `CRTC_X`, `CRTC_Y`, `CRTC_W`, and `CRTC_H`. The primary plane should also have the color encoding and color range properties set based on the source content:

- `COLOR_RANGE`, `COLOR_ENCODING`

The overlay plane should be the native size of the CRTC. The compositor must draw a transparent cutout for where the video should be placed on the desktop (i.e., set the alpha to zero). The primary plane video will be visible through the underlay. The overlay plane’s buffer may remain static while the primary plane’s framebuffer is used for standard double-buffered playback.

The compositor should create a YUV buffer matching the native size of the CRTC. Each video buffer should be composited onto this YUV buffer for direct YUV scanout. The primary plane should have the color encoding and color range properties set based on the source content: `COLOR_RANGE`, `COLOR_ENCODING`. However, be mindful that the source color space and encoding match for each video since it affect the entire plane.

The overlay plane should be the native size of the CRTC. The compositor must draw a transparent cutout for where each video should be placed on the desktop (i.e., set the alpha to zero).

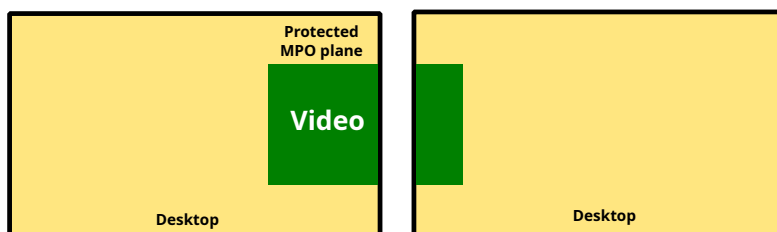
The primary plane videos will be visible through the underlay. The overlay plane's buffer may remain static while compositing operations for video playback will be done on the video buffer.

This kernel interface is validated using IGT GPU Tools. The following tests can be run to validate positioning, blending, scaling under a variety of sequences and interactions with operations such as DPMS and S3:

- kms_plane@plane-panning-bottom-right-pipe-*-planes
- kms_plane@plane-panning-bottom-right-suspend-pipe-*-
- kms_plane@plane-panning-top-left-pipe-*-
- kms_plane@plane-position-covered-pipe-*-
- kms_plane@plane-position-hole-dpms-pipe-*-
- kms_plane@plane-position-hole-pipe-*-
- kms_plane_multiple@atomic-pipe-*-tiling-
- kms_plane_scaling@pipe-*-plane-scaling
- kms_plane_alpha_blend@pipe-*-alpha-basic
- kms_plane_alpha_blend@pipe-*-alpha-transparent-fb
- kms_plane_alpha_blend@pipe-*-alpha-opaque-fb
- kms_plane_alpha_blend@pipe-*-constant-alpha-min
- kms_plane_alpha_blend@pipe-*-constant-alpha-mid
- kms_plane_alpha_blend@pipe-*-constant-alpha-max

Multiple Display MPO

AMDGPU supports display MPO when using multiple displays; however, this feature behavior heavily relies on the compositor implementation. Keep in mind that userspace can define different policies. For example, some OSes can use MPO to protect the plane that handles the video playback; notice that we don't have many limitations for a single display. Nonetheless, this manipulation can have many more restrictions for a multi-display scenario. The below example shows a video playback in the middle of two displays, and it is up to the compositor to define a policy on how to handle it:



Let's discuss some of the hardware limitations we have when dealing with multi-display with MPO.

Limitations

For simplicity's sake, for discussing the hardware limitation, this documentation supposes an example where we have two displays and video playback that will be moved around different displays.

- **Hardware limitations**

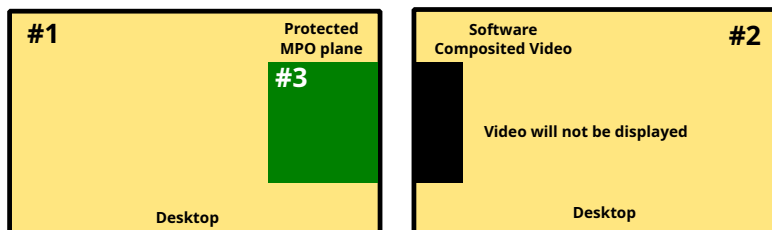
From the DCN overview page, each display requires at least one pipe and each MPO plane needs another pipe. As a result, when the video is in the middle of the two displays, we need to use 2 pipes. See the example below where we avoid pipe split:

- 1 display (1 pipe) + MPO (1 pipe), we will use two pipes
- 2 displays (2 pipes) + MPO (1-2 pipes); we will use 4 pipes. MPO in the middle of both displays needs 2 pipes.
- 3 Displays (3 pipes) + MPO (1-2 pipes), we need 5 pipes.

If we use MPO with multiple displays, the userspace has to decide to enable multiple MPO by the price of limiting the number of external displays supported or disable it in favor of multiple displays; it is a policy decision. For example:

- When ASIC has 3 pipes, AMD hardware can NOT support 2 displays with MPO
- When ASIC has 4 pipes, AMD hardware can NOT support 3 displays with MPO

Let's briefly explore how userspace can handle these two display configurations on an ASIC that only supports three pipes. We can have:



- Total pipes are 3
 - User lights up 2 displays (2 out of 3 pipes are used)
 - User launches video (1 pipe used for MPO)
 - Now, if the user moves the video in the middle of 2 displays, one part of the video won't be MPO since we have used 3/3 pipes.
- **Scaling limitation**

MPO cannot handle scaling less than 0.25 and more than x16. For example:

If 4k video (3840x2160) is playing in windowed mode, the physical size of the window cannot be smaller than (960x540).

Note: These scaling limitations might vary from ASIC to ASIC.

- **Size Limitation**

The minimum MPO size is 12px.

DC Glossary

On this page, we try to keep track of acronyms related to the display component. If you do not find what you are looking for, look at the '[AMDGPU Glossary](#)'; if you cannot find it anywhere, consider asking in the amdgfx and update this page.

ABM

Adaptive Backlight Modulation

APU

Accelerated Processing Unit

ASIC

Application-Specific Integrated Circuit

ASSR

Alternate Scrambler Seed Reset

AZ

Azalia (HD audio DMA engine)

BPC

Bits Per Colour/Component

BPP

Bits Per Pixel

Clocks

- PCLK: Pixel Clock
- SYMCLK: Symbol Clock
- SOCCLK: GPU Engine Clock
- DISPCLK: Display Clock
- DPPCLK: DPP Clock
- DCFCLK: Display Controller Fabric Clock
- REFCLK: Real Time Reference Clock
- PPLL: Pixel PLL
- FCLK: Fabric Clock
- MCLK: Memory Clock

CRC

Cyclic Redundancy Check

CRTC

Cathode Ray Tube Controller - commonly called "Controller" - Generates raw stream of pixels, clocked at pixel clock

CVT

Coordinated Video Timings

DAL

Display Abstraction layer

DC (Software)

Display Core

DC (Hardware)

Display Controller

DCC

Delta Colour Compression

DCE

Display Controller Engine

DCHUB

Display Controller HUB

ARB

Arbiter

VTG

Vertical Timing Generator

DCN

Display Core Next

DCCG

Display Clock Generator block

DDC

Display Data Channel

DIO

Display IO

DPP

Display Pipes and Planes

DSC

Display Stream Compression (Reduce the amount of bits to represent pixel count while at the same pixel clock)

dGPU

discrete GPU

DMIF

Display Memory Interface

DML

Display Mode Library

DMCU

Display Micro-Controller Unit

DMCUB

Display Micro-Controller Unit, version B

DPCD

DisplayPort Configuration Data

DPM(S)

Display Power Management (Signaling)

DRR

Dynamic Refresh Rate

DWB

Display Writeback

FB

Frame Buffer

FBC

Frame Buffer Compression

FEC

Forward Error Correction

FRL

Fixed Rate Link

GCO

Graphical Controller Object

GSL

Global Swap Lock

iGPU

integrated GPU

ISR

Interrupt Service Request

ISV

Independent Software Vendor

KMD

Kernel Mode Driver

LB

Line Buffer

LFC

Low Framerate Compensation

LTTPR

Link Training Tunable Phy Repeater

LUT

Lookup Table

MALL

Memory Access at Last Level

MC

Memory Controller

MPC/MPCC

Multiple pipes and plane combine

MPO

Multi Plane Overlay

MST

Multi Stream Transport

NBP State

Northbridge Power State

NBIO

North Bridge Input/Output

ODM

Output Data Mapping

OPM

Output Protection Manager

OPP

Output Plane Processor

OPTC

Output Pipe Timing Combiner

OTG

Output Timing Generator

PCON

Power Controller

PGFSM

Power Gate Finite State Machine

PSR

Panel Self Refresh

SCL

Scaler

SDP

Scalable Data Port

SLS

Single Large Surface

SST

Single Stream Transport

TMDS

Transition-Minimized Differential Signaling

TMZ

Trusted Memory Zone

TTU

Time to Underflow

VRR

Variable Refresh Rate

UVD

Unified Video Decoder

10.1.4 dGPU firmware flashing

IFWI

Flashing the dGPU integrated firmware image (IFWI) is supported by GPUs that use the PSP to orchestrate the update (Navi3x or newer GPUs). For supported GPUs, *amdgpu* will export a series of sysfs files that can be used for the flash process.

The IFWI flash process is:

1. Ensure the IFWI image is intended for the dGPU on the system.
2. “Write” the IFWI image to the sysfs file *psp_vbflash*. This will stage the IFWI in memory.
3. “Read” from the *psp_vbflash* sysfs file to initiate the flash process.
4. Poll the *psp_vbflash_status* sysfs file to determine when the flash process completes.

USB-C PD F/W

On GPUs that support flashing an updated USB-C PD firmware image, the process is done using the *usbc_pd_fw* sysfs file.

- Reading the file will provide the current firmware version.
- Writing the name of a firmware payload stored in */lib/firmware/amdgpu* to the sysfs file will initiate the flash process.

The firmware payload stored in */lib/firmware/amdgpu* can be named any name as long as it doesn’t conflict with other existing binaries that are used by *amdgpu*.

sysfs files

usbc_pd_fw

Reading from this file will retrieve the USB-C PD firmware version. Writing to this file will trigger the update process.

psp_vbflash

Writing to this file will stage an IFWI for update. Reading from this file will trigger the update process.

psp_vbflash_status

The status of the flash process. 0: IFWI flash not complete. 1: IFWI flash complete.

10.1.5 AMDGPU XGMI Support

AMDGPU XGMI Support

XGMI is a high speed interconnect that joins multiple GPU cards into a homogeneous memory space that is organized by a collective hive ID and individual node IDs, both of which are 64-bit numbers.

The file `xgmi_device_id` contains the unique per GPU device ID and is stored in the `/sys/class/drm/card${cardno}/device/` directory.

Inside the device directory a sub-directory 'xgmi_hive_info' is created which contains the hive ID and the list of nodes.

The hive ID is stored in:

`/sys/class/drm/card${cardno}/device/xgmi_hive_info/xgmi_hive_id`

The node information is stored in numbered directories:

`/sys/class/drm/card${cardno}/device/xgmi_hive_info/node${nodeno}/xgmi_device_id`

Each device has their own `xgmi_hive_info` direction with a mirror set of node sub-directories.

The XGMI memory space is built by contiguously adding the power of two padded VRAM space from each node to each other.

10.1.6 AMDGPU RAS Support

The AMDGPU RAS interfaces are exposed via `sysfs` (for informational queries) and `debugfs` (for error injection).

RAS debugfs/sysfs Control and Error Injection Interfaces

The control interface accepts struct `ras_debug_if` which has two members.

First member: `ras_debug_if::head` or `ras_debug_if::inject`.

`head` is used to indicate which IP block will be under control.

`head` has four members, they are `block`, `type`, `sub_block_index`, `name`. `block`: which IP will be under control. `type`: what kind of error will be enabled/disabled/injected. `sub_block_index`: some IPs have subcomponents. say, GFX, sDMA. `name`: the name of IP.

`inject` has three more members than `head`, they are `address`, `value` and `mask`. As their names indicate, `inject` operation will write the value to the address.

The second member: struct `ras_debug_if::op`. It has three kinds of operations.

- 0: disable RAS on the block. Take `::head` as its data.
- 1: enable RAS on the block. Take `::head` as its data.
- 2: inject errors on the block. Take `::inject` as its data.

How to use the interface?

In a program

Copy the struct `ras_debug_if` in your code and initialize it. Write the struct to the control interface.

From shell

```
echo "disable <block>" > /sys/kernel/debug/dri/<N>/ras/ras_ctrl
echo "enable <block> <error>" > /sys/kernel/debug/dri/<N>/ras/ras_ctrl
echo "inject <block> <error> <sub-block> <address> <value> <mask>" > /sys/
kernel/debug/dri/<N>/ras/ras_ctrl
```

Where N, is the card which you want to affect.

“disable” requires only the block. “enable” requires the block and error type. “inject” requires the block, error type, address, and value.

The block is one of: umc, sdma, gfx, etc.

see ras_block_string[] for details

The error type is one of: ue, ce, where,

ue is multi-uncorrectable ce is single-correctable

The sub-block is a the sub-block index, pass 0 if there is no sub-block. The address and value are hexadecimal numbers, leading 0x is optional. The mask means instance mask, is optional, default value is 0x1.

For instance,

```
echo inject umc ue 0x0 0x0 0x0 > /sys/kernel/debug/dri/0/ras/ras_ctrl
echo inject umc ce 0 0 0 3 > /sys/kernel/debug/dri/0/ras/ras_ctrl
echo disable umc > /sys/kernel/debug/dri/0/ras/ras_ctrl
```

How to check the result of the operation?

To check disable/enable, see “ras” features at, /sys/class/drm/card[0/1/2...]/device/ras/features

To check inject, see the corresponding error count at, /sys/class/drm/card[0/1/2...]/device/ras/[gfx|sdma]

Note: Operations are only allowed on blocks which are supported. Check the “ras” mask at /sys/module/amdgpu/parameters/ras_mask to see which blocks support RAS on a particular asic.

RAS Reboot Behavior for Unrecoverable Errors

Normally when there is an uncorrectable error, the driver will reset the GPU to recover. However, in the event of an unrecoverable error, the driver provides an interface to reboot the system automatically in that event.

The following file in debugfs provides that interface: /sys/kernel/debug/dri/[0/1/2...]/ras/auto_reboot

Usage:

```
echo true > .../ras/auto_reboot
```

RAS Error Count sysfs Interface

It allows the user to read the error count for each IP block on the gpu through `/sys/class/drm/card[0/1/2...]/device/ras/[gfx/sdma/...]_err_count`

It outputs the multiple lines which report the uncorrected (ue) and corrected (ce) error counts.

The format of one line is below,

`[ce|ue]: count`

Example:

```
ue: 0
ce: 1
```

RAS EEPROM debugfs Interface

Some boards contain an EEPROM which is used to persistently store a list of bad pages which experiences ECC errors in vram. This interface provides a way to reset the EEPROM, e.g., after testing error injection.

Usage:

```
echo 1 > ../ras/ras_eeprom_reset
```

will reset EEPROM table to 0 entries.

RAS VRAM Bad Pages sysfs Interface

It allows user to read the bad pages of vram on the gpu through `/sys/class/drm/card[0/1/2...]/device/ras/gpu_vram_bad_pages`

It outputs multiple lines, and each line stands for one gpu page.

The format of one line is below, gpu pfn : gpu page size : flags

gpu pfn and gpu page size are printed in hex format. flags can be one of below character,

R: reserved, this gpu page is reserved and not able to use.

P: pending for reserve, this gpu page is marked as bad, will be reserved in next window of `page_reserve`.

F: unable to reserve. this gpu page can't be reserved due to some reasons.

Examples:

```
0x00000001 : 0x00001000 : R
0x00000002 : 0x00001000 : P
```


Sample Code

Sample code for testing error injection can be found here: https://cgit.freedesktop.org/mesa/drm/tree/tests/amdgpu/ras_tests.c

This is part of the libdrm amdgpu unit tests which cover several areas of the GPU. There are four sets of tests:

RAS Basic Test

The test verifies the RAS feature enabled status and makes sure the necessary sysfs and debugfs files are present.

RAS Query Test

This test checks the RAS availability and enablement status for each supported IP block as well as the error counts.

RAS Inject Test

This test injects errors for each IP.

RAS Disable Test

This test tests disabling of RAS features for each IP block.

10.1.7 GPU Power/Thermal Controls and Monitoring

HWMON Interfaces

The amdgpu driver exposes the following sensor interfaces:

- GPU temperature (via the on-die sensor)
- GPU voltage
- Northbridge voltage (APUs only)
- GPU power
- GPU fan
- GPU gfx/compute engine clock
- GPU memory clock (dGPU only)

hwmon interfaces for GPU temperature:

- temp[1-3]_input: the on die GPU temperature in millidegrees Celsius - temp2_input and temp3_input are supported on SOC15 dGPUs only
- temp[1-3]_label: temperature channel label - temp2_label and temp3_label are supported on SOC15 dGPUs only
- temp[1-3]_crit: temperature critical max value in millidegrees Celsius - temp2_crit and temp3_crit are supported on SOC15 dGPUs only
- temp[1-3]_crit_hyst: temperature hysteresis for critical limit in millidegrees Celsius - temp2_crit_hyst and temp3_crit_hyst are supported on SOC15 dGPUs only

- temp[1-3]_emergency: temperature emergency max value(asic shutdown) in millidegrees Celsius - these are supported on SOC15 dGPUs only

hwmon interfaces for GPU voltage:

- in0_input: the voltage on the GPU in millivolts
- in1_input: the voltage on the Northbridge in millivolts

hwmon interfaces for GPU power:

- power1_average: average power used by the SoC in microWatts. On APUs this includes the CPU.
- power1_input: instantaneous power used by the SoC in microWatts. On APUs this includes the CPU.
- power1_cap_min: minimum cap supported in microWatts
- power1_cap_max: maximum cap supported in microWatts
- power1_cap: selected power cap in microWatts

hwmon interfaces for GPU fan:

- pwm1: pulse width modulation fan level (0-255)
- pwm1_enable: pulse width modulation fan control method (0: no fan speed control, 1: manual fan speed control using pwm interface, 2: automatic fan speed control)
- pwm1_min: pulse width modulation fan control minimum level (0)
- pwm1_max: pulse width modulation fan control maximum level (255)
- fan1_min: a minimum value Unit: revolution/min (RPM)
- fan1_max: a maximum value Unit: revolution/max (RPM)
- fan1_input: fan speed in RPM
- fan[1-*]_target: Desired fan speed Unit: revolution/min (RPM)
- fan[1-*]_enable: Enable or disable the sensors.1: Enable 0: Disable

NOTE: DO NOT set the fan speed via “pwm1” and “fan[1-*]_target” interfaces at the same time.

That will get the former one overridden.

hwmon interfaces for GPU clocks:

- freq1_input: the gfx/compute clock in hertz
- freq2_input: the memory clock in hertz

You can use hwmon tools like sensors to view this information on your system.

GPU sysfs Power State Interfaces

GPU power controls are exposed via sysfs files.

power_dpm_state

The `power_dpm_state` file is a legacy interface and is only provided for backwards compatibility. The `amdgpu` driver provides a sysfs API for adjusting certain power related parameters. The file `power_dpm_state` is used for this. It accepts the following arguments:

- battery
- balanced
- performance

battery

On older GPUs, the vbios provided a special power state for battery operation. Selecting battery switched to this state. This is no longer provided on newer GPUs so the option does nothing in that case.

balanced

On older GPUs, the vbios provided a special power state for balanced operation. Selecting balanced switched to this state. This is no longer provided on newer GPUs so the option does nothing in that case.

performance

On older GPUs, the vbios provided a special power state for performance operation. Selecting performance switched to this state. This is no longer provided on newer GPUs so the option does nothing in that case.

power_dpm_force_performance_level

The `amdgpu` driver provides a sysfs API for adjusting certain power related parameters. The file `power_dpm_force_performance_level` is used for this. It accepts the following arguments:

- auto
- low
- high
- manual
- profile_standard
- profile_min_sclk
- profile_min_mclk
- profile_peak

auto

When auto is selected, the driver will attempt to dynamically select the optimal power profile for current conditions in the driver.

low

When low is selected, the clocks are forced to the lowest power state.

high

When high is selected, the clocks are forced to the highest power state.

manual

When manual is selected, the user can manually adjust which power states are enabled for each clock domain via the sysfs `pp_dpm_mclk`, `pp_dpm_sclk`, and `pp_dpm_pcie` files and adjust the power state transition heuristics via the `pp_power_profile_mode` sysfs file.

`profile_standard` `profile_min_sclk` `profile_min_mclk` `profile_peak`

When the profiling modes are selected, clock and power gating are disabled and the clocks are set for different profiling cases. This mode is recommended for profiling specific work loads where you do not want clock or power gating for clock fluctuation to interfere with your results. `profile_standard` sets the clocks to a fixed clock level which varies from asic to asic. `profile_min_sclk` forces the `sclk` to the lowest level. `profile_min_mclk` forces the `mclk` to the lowest level. `profile_peak` sets all clocks (`mclk`, `sclk`, `pcie`) to the highest levels.

pp_table

The `amdgpu` driver provides a sysfs API for uploading new powerplay tables. The file `pp_table` is used for this. Reading the file will dump the current power play table. Writing to the file will attempt to upload a new powerplay table and re-initialize powerplay using that new table.

pp_od_clk_voltage

The `amdgpu` driver provides a sysfs API for adjusting the clocks and voltages in each power level within a power state. The `pp_od_clk_voltage` is used for this.

Note that the actual memory controller clock rate are exposed, not the effective memory clock of the DRAMs. To translate it, use the following formula:

Clock conversion (Mhz):

HBM: $\text{effective_memory_clock} = \text{memory_controller_clock} * 1$

G5: $\text{effective_memory_clock} = \text{memory_controller_clock} * 1$

G6: $\text{effective_memory_clock} = \text{memory_controller_clock} * 2$

DRAM data rate (MT/s):

HBM: $\text{effective_memory_clock} * 2 = \text{data_rate}$

G5: $\text{effective_memory_clock} * 4 = \text{data_rate}$

G6: $\text{effective_memory_clock} * 8 = \text{data_rate}$

Bandwidth (MB/s):

$\text{data_rate} * \text{vram_bit_width} / 8 = \text{memory_bandwidth}$

Some examples:

G5 on RX460:

`memory_controller_clock = 1750 Mhz`

`effective_memory_clock = 1750 Mhz * 1 = 1750 Mhz`

`data rate = 1750 * 4 = 7000 MT/s`

`memory_bandwidth = 7000 * 128 bits / 8 = 112000 MB/s`

G6 on RX5700:

`memory_controller_clock = 875 Mhz`

`effective_memory_clock = 875 Mhz * 2 = 1750 Mhz`

`data rate = 1750 * 8 = 14000 MT/s`

`memory_bandwidth = 14000 * 256 bits / 8 = 448000 MB/s`

< For Vega10 and previous ASICs >

Reading the file will display:

- a list of engine clock levels and voltages labeled OD_SCLK
- a list of memory clock levels and voltages labeled OD_MCLK
- a list of valid ranges for sclk, mclk, and voltage labeled OD_RANGE

To manually adjust these settings, first select manual using `power_dpm_force_performance_level`. Enter a new value for each level by writing a string that contains "s/m level clock voltage" to the file. E.g., "s 1 500 820" will update sclk level 1 to be 500 MHz at 820 mV; "m 0 350 810" will update mclk level 0 to be 350 MHz at 810 mV. When you have edited all of the states as needed, write "c" (commit) to the file to commit your changes. If you want to reset to the default power levels, write "r" (reset) to the file to reset them.

< For Vega20 and newer ASICs >

Reading the file will display:

- minimum and maximum engine clock labeled OD_SCLK
- minimum(not available for Vega20 and Navi1x) and maximum memory clock labeled OD_MCLK
- three <frequency, voltage> points labeled OD_VDDC_CURVE. They can be used to calibrate the sclk voltage curve. This is available for Vega20 and NV1X.
- voltage offset for the six anchor points of the v/f curve labeled OD_VDDC_CURVE. They can be used to calibrate the v/f curve. This is only available for some SMU13 ASICs.
- voltage offset(in mV) applied on target voltage calculation. This is available for Sienna Cichlid, Navy Flounder and Dimgrey Cavefish. For these ASICs, the target voltage calculation can be illustrated by "voltage = voltage calculated from v/f curve + overdrive vddgfx offset"
- a list of valid ranges for sclk, mclk, and voltage curve points labeled OD_RANGE

< For APU's >

Reading the file will display:

- minimum and maximum engine clock labeled OD_SCLK
- a list of valid ranges for sclk labeled OD_RANGE

< For VanGogh >

Reading the file will display:

- minimum and maximum engine clock labeled OD_SCLK
- minimum and maximum core clocks labeled OD_CCLK
- a list of valid ranges for sclk and cclk labeled OD_RANGE

To manually adjust these settings:

- First select manual using `power_dpm_force_performance_level`
- For clock frequency setting, enter a new value by writing a string that contains “s/m index clock” to the file. The index should be 0 if to set minimum clock. And 1 if to set maximum clock. E.g., “s 0 500” will update minimum sclk to be 500 MHz. “m 1 800” will update maximum mclk to be 800Mhz. For core clocks on VanGogh, the string contains “p core index clock”. E.g., “p 2 0 800” would set the minimum core clock on core 2 to 800Mhz.

For sclk voltage curve,

- For NV1X, enter the new values by writing a string that contains “vc point clock voltage” to the file. The points are indexed by 0, 1 and 2. E.g., “vc 0 300 600” will update point1 with clock set as 300Mhz and voltage as 600mV. “vc 2 1000 1000” will update point3 with clock set as 1000Mhz and voltage 1000mV.
- For SMU13 ASICs, enter the new values by writing a string that contains “vc anchor_point_index voltage_offset” to the file. There are total six anchor points defined on the v/f curve with index as 0 - 5. - “vc 0 10” will update the voltage offset for point1 as 10mv. - “vc 5 -10” will update the voltage offset for point6 as -10mv.

To update the voltage offset applied for gfxclk/voltage calculation, enter the new value by writing a string that contains “vo offset”. This is supported by Sienna Cichlid, Navy Flounder and Dimgrey Cavefish. And the offset can be a positive or negative value.

- When you have edited all of the states as needed, write “c” (commit) to the file to commit your changes
- If you want to reset to the default power levels, write “r” (reset) to the file to reset them

pp_dpm_*

The amdgpu driver provides a sysfs API for adjusting what power levels are enabled for a given power state. The files `pp_dpm_sclk`, `pp_dpm_mclk`, `pp_dpm_socclk`, `pp_dpm_fclk`, `pp_dpm_dcefclk` and `pp_dpm_pcie` are used for this.

`pp_dpm_socclk` and `pp_dpm_dcefclk` interfaces are only available for Vega10 and later ASICs. `pp_dpm_fclk` interface is only available for Vega20 and later ASICs.

Reading back the files will show you the available power levels within the power state and the clock information for those levels.

To manually adjust these states, first select manual using `power_dpm_force_performance_level`. Secondly, enter a new value for each level by inputting a string that contains " `echo xx xx xx > pp_dpm_sclk/mclk/pcie`" E.g.,

```
echo "4 5 6" > pp_dpm_sclk
```

will enable sclk levels 4, 5, and 6.

NOTE: change to the dcefclk max dpm level is not supported now

pp_power_profile_mode

The amdgpu driver provides a sysfs API for adjusting the heuristics related to switching between power levels in a power state. The file `pp_power_profile_mode` is used for this.

Reading this file outputs a list of all of the predefined power profiles and the relevant heuristics settings for that profile.

To select a profile or create a custom profile, first select manual using `power_dpm_force_performance_level`. Writing the number of a predefined profile to `pp_power_profile_mode` will enable those heuristics. To create a custom set of heuristics, write a string of numbers to the file starting with the number of the custom profile along with a setting for each heuristic parameter. Due to differences across asic families the heuristic parameters vary from family to family.

***_busy_percent**

The amdgpu driver provides a sysfs API for reading how busy the GPU is as a percentage. The file `gpu_busy_percent` is used for this. The SMU firmware computes a percentage of load based on the aggregate activity level in the IP cores.

The amdgpu driver provides a sysfs API for reading how busy the VRAM is as a percentage. The file `mem_busy_percent` is used for this. The SMU firmware computes a percentage of load based on the aggregate activity level in the IP cores.

gpu_metrics

The amdgpu driver provides a sysfs API for retrieving current gpu metrics data. The file `gpu_metrics` is used for this. Reading the file will dump all the current gpu metrics data.

These data include temperature, frequency, engines utilization, power consume, throttler status, fan speed and cpu core statistics(available for APU only). That's it will give a snapshot of all sensors at the same time.

GFXOFF

GFXOFF is a feature found in most recent GPUs that saves power at runtime. The card's RLC (RunList Controller) firmware powers off the gfx engine dynamically when there is no workload on gfx or compute pipes. GFXOFF is on by default on supported GPUs.

Userspace can interact with GFXOFF through a debugfs interface (all values in *uint32_t*, unless otherwise noted):

amdgpu_gfxoff

Use it to enable/disable GFXOFF, and to check if it's current enabled/disabled:

```
$ xxd -l1 -p /sys/kernel/debug/dri/0/amdgpu_gfxoff
01
```

- Write 0 to disable it, and 1 to enable it.
- Read 0 means it's disabled, 1 it's enabled.

If it's enabled, that means that the GPU is free to enter into GFXOFF mode as needed. Disabled means that it will never enter GFXOFF mode.

amdgpu_gfxoff_status

Read it to check current GFXOFF's status of a GPU:

```
$ xxd -l1 -p /sys/kernel/debug/dri/0/amdgpu_gfxoff_status
02
```

- 0: GPU is in GFXOFF state, the gfx engine is powered down.
- 1: Transition out of GFXOFF state
- 2: Not in GFXOFF state
- 3: Transition into GFXOFF state

If GFXOFF is enabled, the value will be transitioning around [0, 3], always getting into 0 when possible. When it's disabled, it's always at 2. Returns -EINVAL if it's not supported.

amdgpu_gfxoff_count

Read it to get the total GFXOFF entry count at the time of query since system power-up. The value is an *uint64_t* type, however, due to firmware limitations, it can currently overflow as an *uint32_t*. *Only supported in vangogh*

amdgpu_gfxoff_residency

Write 1 to `amdgpu_gfxoff_residency` to start logging, and 0 to stop. Read it to get average GFXOFF residency % multiplied by 100 during the last logging interval. E.g. a value of 7854 means 78.54% of the time in the last logging interval the GPU was in GFXOFF mode. *Only supported in vangogh*

10.1.8 Misc AMDGPU driver information

GPU Product Information

Information about the GPU can be obtained on certain cards via sysfs

product_name

The `amdgpu` driver provides a sysfs API for reporting the product name for the device. The file `product_name` is used for this and returns the product name as returned from the FRU. NOTE: This is only available for certain server cards.

product_number

The `amdgpu` driver provides a sysfs API for reporting the part number for the device. The file `product_number` is used for this and returns the part number as returned from the FRU. NOTE: This is only available for certain server cards.

serial_number

The `amdgpu` driver provides a sysfs API for reporting the serial number for the device. The file `serial_number` is used for this and returns the serial number as returned from the FRU. NOTE: This is only available for certain server cards.

unique_id

The `amdgpu` driver provides a sysfs API for providing a unique ID for the GPU. The file `unique_id` is used for this. This will provide a Unique ID that will persist from machine to machine.

NOTE: This will only work for GFX9 and newer. This file will be absent on unsupported ASICs (GFX8 and older).

Accelerated Processing Units (APU) Info

Product Name	Code ence	Refer-	DCN/DCE sion	ver-	GC ver- sion	VCE/UV ver- sion	SDMA ver- sion	MP0 ver- sion
Radeon R* Graphics	CAR- RIZO/STONE		DCE 11		8	VCE 3 / UVD 6	3	n/a
Ryzen 3000 series / AMD Ryzen Embedded V1*/R1* with Radeon Vega Gfx	RAVEN/PICASSO		DCN 1.0		9.1.0	VCN 1.0	4.1.0	10.0.0
Ryzen 4000 series	RENOIR		DCN 2.1		9.3	VCN 2.2	4.1.2	11.0.3
Ryzen 3000 series / AMD Ryzen Embedded V1*/R1* with Radeon Vega Gfx	RAVEN2		DCN 1.0		9.2.2	VCN 1.0.1	4.1.1	10.0.1
SteamDeck	VANGOGH		DCN 3.0.1		10.3.1	VCN 3.1.0	5.2.1	11.5.0
Ryzen 5000 series / Ryzen 7x30 series	GREEN SARDINE / Cezanne / Barcelo / Barcelo-R		DCN 2.1		9.3	VCN 2.2	4.1.1	12.0.1
Ryzen 6000 series / Ryzen 7x35 series / Ryzen 7x36 series	YELLOW CARP / Rem- brandt / Rembrandt- R		3.1.2		10.3.3	VCN 3.1.1	5.2.3	13.0.3
Ryzen 7000 series (AM5)	Raphael		3.1.5		10.3.6	3.1.2	5.2.6	13.0.5
Ryzen 7x45 series (FL1)	/ Dragon Range		3.1.5		10.3.6	3.1.2	5.2.6	13.0.5
Ryzen 7x20 series	Mendocino		3.1.6		10.3.7	3.1.1	5.2.7	13.0.8
Ryzen 7x40 series	Phoenix		3.1.4		11.0.1 / 11.0.4	4.0.2	6.0.1	13.0.4 / 13.0.11

Discrete GPU Info

Product Name	Code Reference	DCN/DCE sion	ver-	GC ver- sion	VCN version	SDMA version
AMD Radeon (TM) HD 8500M/ 8600M /M200 /M320 /M330 /M335 Series	HAINAN	--		6	--	--
AMD Radeon HD 7800 /7900 /FireGL Series	TAHITI	DCE 6		6	VCE 1 / UVD 3	--
AMD Radeon R7 (TM HD) M265 /M370 /8500M /8600 /8700 /8700M	OLAND	DCE 6		6	VCE 1 / UVD 3	--
AMD Radeon (TM) (HD R7) 7800 /7970 /8800 /8970 /370/ Series	PITCAIRN	DCE 6		6	VCE 1 / UVD 3	--
AMD Radeon (TM R7 R9 HD) E8860 /M360 /7700 /7800 /8800 /9000(M) /W4100 Series	VERDE	DCE 6		6	VCE 1 / UVD 3	--
AMD Radeon HD M280X /M380 /7700 /8950 /W5100	BONAIRE	DCE 8		7	VCE 2 / UVD 4.2	1
AMD Radeon (R9 TM) 200 /390 /W8100 /W9100 Series	HAWAII	DCE 8		7	VCE 2 / UVD 4.2	1
AMD Radeon (TM) R(5 7) M315 /M340 /M360	TOPAZ	.		8	--	2
AMD Radeon (TM) R9 200 /380 /W7100 /S7150 /M390 /M395 Series	TONGA	DCE 10		8	VCE 3 / UVD 5	3
AMD Radeon (FirePro) (TM) R9 Fury Series	FIJI	DCE 10		8	VCE 3 / UVD 6	3
Radeon RX 470 /480 /570 /580 /590 Series - AMD Radeon (TM) (Pro WX) 5100 /E9390 /E9560 /E9565 /V7350 /7100 /P30PH	POLARIS10	DCE 11.2		8	VCE 3.4 / UVD 6.3	3
Radeon (TM) (RX Pro WX) E9260 /460 /V5300X /550 /560(X) Series	POLARIS11	DCE 11.2		8	VCE 3.4 / UVD 6.3	3
Radeon (RX/Pro) 500 /540(X) /550 /640 /WX2100 /WX3100 /WX200 Series	POLARIS12	DCE 11.2		8	VCE 3.4 / UVD 6.3	3
Radeon (RX TM) (PRO WX) Vega /MI25 /V320 /V340L /8200 /9100 /SSG MxGPU	VEGA10	DCE 12		9.0.1	VCE 4.0.0 / UVD 7.0.0	4.0.0
AMD Radeon (Pro) VII /MI50 /MI60	VEGA20	DCE 12		9.4.0	VCE 4.1.0 / UVD 7.2.0	4.2.0

GPU Memory Usage Information

Various memory accounting can be accessed via sysfs

mem_info_vram_total

The amdgpu driver provides a sysfs API for reporting current total VRAM available on the device. The file `mem_info_vram_total` is used for this and returns the total amount of VRAM in bytes.

mem_info_vram_used

The amdgpu driver provides a sysfs API for reporting current total VRAM available on the device. The file `mem_info_vram_used` is used for this and returns the total amount of currently used VRAM in bytes.

mem_info_vis_vram_total

The amdgpu driver provides a sysfs API for reporting current total visible VRAM available on the device. The file `mem_info_vis_vram_total` is used for this and returns the total amount of visible VRAM in bytes.

mem_info_vis_vram_used

The amdgpu driver provides a sysfs API for reporting current total of used visible VRAM. The file `mem_info_vis_vram_used` is used for this and returns the total amount of currently used visible VRAM in bytes.

mem_info_gtt_total

The amdgpu driver provides a sysfs API for reporting current total size of the GTT. The file `mem_info_gtt_total` is used for this, and returns the total size of the GTT block, in bytes.

mem_info_gtt_used

The amdgpu driver provides a sysfs API for reporting current total amount of used GTT. The file `mem_info_gtt_used` is used for this, and returns the current used size of the GTT block, in bytes.

PCIe Accounting Information

pcie_bw

The amdgpu driver provides a sysfs API for estimating how much data has been received and sent by the GPU in the last second through PCIe. The file `pcie_bw` is used for this. The Perf counters count the number of received and sent messages and return those values, as well as the maximum payload size of a PCIe packet (mps). Note that it is not possible to easily and quickly obtain the size of each packet transmitted, so we output the max payload size (mps) to allow for quick estimation of the PCIe bandwidth usage

pcie_replay_count

The amdgpu driver provides a sysfs API for reporting the total number of PCIe replays (NAKs) The file `pcie_replay_count` is used for this and returns the total number of replays as a sum of the NAKs generated and NAKs received

GPU SmartShift Information

GPU SmartShift information via sysfs

smartshift_apu_power

The amdgpu driver provides a sysfs API for reporting APU power shift in percentage if platform supports smartshift. Value 0 means that there is no powershift and values between [1-100] means that the power is shifted to APU, the percentage of boost is with respect to APU power limit on the platform.

smartshift_dgpu_power

The amdgpu driver provides a sysfs API for reporting dGPU power shift in percentage if platform supports smartshift. Value 0 means that there is no powershift and values between [1-100] means that the power is shifted to dGPU, the percentage of boost is with respect to dGPU power limit on the platform.

smartshift_bias

The amdgpu driver provides a sysfs API for reporting the smartshift(SS2.0) bias level. The value ranges from -100 to 100 and the default is 0. -100 sets maximum preference to APU and 100 sets max preference to dGPU.

10.1.9 AMDGPU Glossary

Here you can find some generic acronyms used in the amdgpu driver. Notice that we have a dedicated glossary for Display Core at '[DC Glossary](#)'.

active_cu_number

The number of CUs that are active on the system. The number of active CUs may be less than $SE * SH * CU$ depending on the board configuration.

CP

Command Processor

CPLIB

Content Protection Library

CU

Compute Unit

DFS

Digital Frequency Synthesizer

ECP

Enhanced Content Protection

EOP

End Of Pipe/Pipeline

GART

Graphics Address Remapping Table. This is the name we use for the GPUVM page table used by the GPU kernel driver. It remaps system resources (memory or MMIO space) into the GPU's address space so the GPU can access them. The name GART harkens back to the days of AGP when the platform provided an MMU that the GPU could use to get a contiguous view of scattered pages for DMA. The MMU has since moved on to the GPU, but the name stuck.

GC

Graphics and Compute

GMC

Graphic Memory Controller

GPUVM

GPU Virtual Memory. This is the GPU's MMU. The GPU supports multiple virtual address spaces that can be in flight at any given time. These allow the GPU to remap VRAM and system resources into GPU virtual address spaces for use by the GPU kernel driver and applications using the GPU. These provide memory protection for different applications using the GPU.

GTT

Graphics Translation Tables. This is a memory pool managed through TTM which provides access to system resources (memory or MMIO space) for use by the GPU. These addresses can be mapped into the "GART" GPUVM page table for use by the kernel driver or into per process GPUVM page tables for application usage.

IH

Interrupt Handler

HQD

Hardware Queue Descriptor

IB

Indirect Buffer

IP

Intellectual Property blocks

KCQ

Kernel Compute Queue

KGQ

Kernel Graphics Queue

KIQ

Kernel Interface Queue

MEC

MicroEngine Compute

MES

MicroEngine Scheduler

MMHUB

Multi-Media HUB

MQD

Memory Queue Descriptor

PPLib

PowerPlay Library - PowerPlay is the power management component.

PSP

Platform Security Processor

RLC

RunList Controller

SDMA

System DMA

SE

Shader Engine

SH

SHeader array

SMU

System Management Unit

SS

Spread Spectrum

VCE

Video Compression Engine

VCN

Video Codec Next

10.2 drm/i915 Intel GFX Driver

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

10.2.1 Core Driver Infrastructure

This section covers core driver infrastructure used by both the display and the GEM parts of the driver.

Runtime Power Management

The i915 driver supports dynamic enabling and disabling of entire hardware blocks at runtime. This is especially important on the display side where software is supposed to control many power gates manually on recent hardware, since on the GT side a lot of the power management is done by the hardware. But even there some manual control at the device level is required.

Since i915 supports a diverse set of platforms with a unified codebase and hardware engineers just love to shuffle functionality around between power domains there's a sizeable amount of indirection required. This file provides generic functions to the driver for grabbing and releasing references for abstract power domains. It then maps those to the actual power wells present for a given platform.

`intel_wakeref_t intel_runtime_pm_get_raw(struct intel_runtime_pm *rpm)`
grab a raw runtime pm reference

Parameters

`struct intel_runtime_pm *rpm`
the intel_runtime_pm structure

Description

This is the unlocked version of `intel_display_power_is_enabled()` and should only be used from error capture and recovery code where deadlocks are possible. This function grabs a device-level runtime pm reference (mostly used for asynchronous PM management from display code) and ensures that it is powered up. Raw references are not considered during wakelock assert checks.

Any runtime pm reference obtained by this function must have a symmetric call to `intel_runtime_pm_put_raw()` to release the reference again.

Return

the wakeref cookie to pass to `intel_runtime_pm_put_raw()`, evaluates as True if the wakeref was acquired, or False otherwise.

`intel_wakeref_t intel_runtime_pm_get(struct intel_runtime_pm *rpm)`
grab a runtime pm reference

Parameters

`struct intel_runtime_pm *rpm`
the intel_runtime_pm structure

Description

This function grabs a device-level runtime pm reference (mostly used for GEM code to ensure the GTT or GT is on) and ensures that it is powered up.

Any runtime pm reference obtained by this function must have a symmetric call to `intel_runtime_pm_put()` to release the reference again.

Return

the wakeref cookie to pass to `intel_runtime_pm_put()`

`intel_wakeref_t __intel_runtime_pm_get_if_active(struct intel_runtime_pm *rpm, bool ignore_usecount)`

grab a runtime pm reference if device is active

Parameters

struct intel_runtime_pm *rpm
the intel_runtime_pm structure

bool ignore_usecount
get a ref even if dev->power.usage_count is 0

Description

This function grabs a device-level runtime pm reference if the device is already active and ensures that it is powered up. It is illegal to try and access the HW should `intel_runtime_pm_get_if_active()` report failure.

If **ignore_usecount** is true, a reference will be acquired even if there is no user requiring the device to be powered up (`dev->power.usage_count == 0`). If the function returns false in this case then it's guaranteed that the device's runtime suspend hook has been called already or that it will be called (and hence it's also guaranteed that the device's runtime resume hook will be called eventually).

Any runtime pm reference obtained by this function must have a symmetric call to `intel_runtime_pm_put()` to release the reference again.

Return

the wakeref cookie to pass to `intel_runtime_pm_put()`, evaluates as True if the wakeref was acquired, or False otherwise.

`intel_wakeref_t intel_runtime_pm_get_noresume(struct intel_runtime_pm *rpm)`
grab a runtime pm reference

Parameters

struct intel_runtime_pm *rpm
the intel_runtime_pm structure

Description

This function grabs a device-level runtime pm reference (mostly used for GEM code to ensure the GTT or GT is on).

It will not power up the device but instead only check that it's powered on. Therefore it is only valid to call this functions from contexts where the device is known to be powered up and where trying to power it up would result in hilarity and deadlocks. That pretty much means only

the system suspend/resume code where this is used to grab runtime pm references for delayed setup down in work items.

Any runtime pm reference obtained by this function must have a symmetric call to [intel_runtime_pm_put\(\)](#) to release the reference again.

Return

the wakeref cookie to pass to [intel_runtime_pm_put\(\)](#)

void **intel_runtime_pm_put_raw**(struct intel_runtime_pm *rpm, intel_wakeref_t wref)
release a raw runtime pm reference

Parameters

struct intel_runtime_pm *rpm
the intel_runtime_pm structure

intel_wakeref_t wref
wakeref acquired for the reference that is being released

Description

This function drops the device-level runtime pm reference obtained by [intel_runtime_pm_get_raw\(\)](#) and might power down the corresponding hardware block right away if this is the last reference.

void **intel_runtime_pm_put_unchecked**(struct intel_runtime_pm *rpm)
release an unchecked runtime pm reference

Parameters

struct intel_runtime_pm *rpm
the intel_runtime_pm structure

Description

This function drops the device-level runtime pm reference obtained by [intel_runtime_pm_get\(\)](#) and might power down the corresponding hardware block right away if this is the last reference.

This function exists only for historical reasons and should be avoided in new code, as the correctness of its use cannot be checked. Always use [intel_runtime_pm_put\(\)](#) instead.

void **intel_runtime_pm_put**(struct intel_runtime_pm *rpm, intel_wakeref_t wref)
release a runtime pm reference

Parameters

struct intel_runtime_pm *rpm
the intel_runtime_pm structure

intel_wakeref_t wref
wakeref acquired for the reference that is being released

Description

This function drops the device-level runtime pm reference obtained by [intel_runtime_pm_get\(\)](#) and might power down the corresponding hardware block right away if this is the last reference.

void **intel_runtime_pm_enable**(struct intel_runtime_pm *rpm)
enable runtime pm

Parameters

struct intel_runtime_pm *rpm
the intel_runtime_pm structure

Description

This function enables runtime pm at the end of the driver load sequence.

Note that this function does currently not enable runtime pm for the subordinate display power domains. That is done by `intel_power_domains_enable()`.

void **intel_uncore_forcewake_get**(struct intel_uncore *uncore, enum forcewake_domains fw_domains)
grab forcewake domain references

Parameters

struct intel_uncore *uncore
the intel_uncore structure

enum forcewake_domains fw_domains
forcewake domains to get reference on

Description

This function can be used get GT's forcewake domain references. Normal register access will handle the forcewake domains automatically. However if some sequence requires the GT to not power down a particular forcewake domains this function should be called at the beginning of the sequence. And subsequently the reference should be dropped by symmetric call to `intel_uncore_forcewake_put()`. Usually caller wants all the domains to be kept awake so the **fw_domains** would be then `FORCEWAKE_ALL`.

void **intel_uncore_forcewake_user_get**(struct intel_uncore *uncore)
claim forcewake on behalf of userspace

Parameters

struct intel_uncore *uncore
the intel_uncore structure

Description

This function is a wrapper around `intel_uncore_forcewake_get()` to acquire the GT powerwell and in the process disable our debugging for the duration of userspace's bypass.

void **intel_uncore_forcewake_user_put**(struct intel_uncore *uncore)
release forcewake on behalf of userspace

Parameters

struct intel_uncore *uncore
the intel_uncore structure

Description

This function complements `intel_uncore_forcewake_user_get()` and releases the GT powerwell taken on behalf of the userspace bypass.

void **intel_uncore_forcewake_get__locked**(struct intel_uncore *uncore, enum forcewake_domains fw_domains)

grab forcewake domain references

Parameters

struct intel_uncore *uncore

the intel_uncore structure

enum forcewake_domains fw_domains

forcewake domains to get reference on

Description

See [intel_uncore_forcewake_get\(\)](#). This variant places the onus on the caller to explicitly handle the dev_priv->uncore.lock spinlock.

void **intel_uncore_forcewake_put**(struct intel_uncore *uncore, enum forcewake_domains fw_domains)

release a forcewake domain reference

Parameters

struct intel_uncore *uncore

the intel_uncore structure

enum forcewake_domains fw_domains

forcewake domains to put references

Description

This function drops the device-level forcewakes for specified domains obtained by [intel_uncore_forcewake_get\(\)](#).

void **intel_uncore_forcewake_flush**(struct intel_uncore *uncore, enum forcewake_domains fw_domains)

flush the delayed release

Parameters

struct intel_uncore *uncore

the intel_uncore structure

enum forcewake_domains fw_domains

forcewake domains to flush

void **intel_uncore_forcewake_put__locked**(struct intel_uncore *uncore, enum forcewake_domains fw_domains)

release forcewake domain references

Parameters

struct intel_uncore *uncore

the intel_uncore structure

enum forcewake_domains fw_domains

forcewake domains to put references

Description

See [intel_uncore_forcewake_put\(\)](#). This variant places the onus on the caller to explicitly handle the `dev_priv->uncore.lock` spinlock.

```
int __intel_wait_for_register_fw(struct intel_uncore *uncore, i915_reg_t reg, u32 mask,
                                u32 value, unsigned int fast_timeout_us, unsigned int
                                slow_timeout_ms, u32 *out_value)
```

wait until register matches expected state

Parameters

struct intel_uncore *uncore

the struct intel_uncore

i915_reg_t reg

the register to read

u32 mask

mask to apply to register value

u32 value

expected value

unsigned int fast_timeout_us

fast timeout in microsecond for atomic/tight wait

unsigned int slow_timeout_ms

slow timeout in millisecond

u32 *out_value

optional placeholder to hold registry value

Description

This routine waits until the target register **reg** contains the expected **value** after applying the **mask**, i.e. it waits until

```
(intel_uncore_read_fw(uncore, reg) & mask) == value
```

Otherwise, the wait will timeout after **slow_timeout_ms** milliseconds. For atomic context **slow_timeout_ms** must be zero and **fast_timeout_us** must be not larger than 20,000 microseconds.

Note that this routine assumes the caller holds forcewake asserted, it is not suitable for very long waits. See `intel_wait_for_register()` if you wish to wait without holding forcewake for the duration (i.e. you expect the wait to be slow).

Return

0 if the register matches the desired condition, or -ETIMEDOUT.

```
int __intel_wait_for_register(struct intel_uncore *uncore, i915_reg_t reg, u32 mask, u32
                                value, unsigned int fast_timeout_us, unsigned int
                                slow_timeout_ms, u32 *out_value)
```

wait until register matches expected state

Parameters

struct intel_uncore *uncore

the struct intel_uncore

i915_reg_t reg

the register to read

u32 mask

mask to apply to register value

u32 value

expected value

unsigned int fast_timeout_us

fast timeout in microsecond for atomic/tight wait

unsigned int slow_timeout_ms

slow timeout in millisecond

u32 *out_value

optional placeholder to hold registry value

Description

This routine waits until the target register **reg** contains the expected **value** after applying the **mask**, i.e. it waits until

```
(intel_uncore_read(uncore, reg) & mask) == value
```

Otherwise, the wait will timeout after **timeout_ms** milliseconds.

Return

0 if the register matches the desired condition, or -ETIMEDOUT.

```
enum forcewake_domains intel_uncore_forcewake_for_reg(struct intel_uncore *uncore,
                                                    i915_reg_t reg, unsigned int
                                                    op)
```

which forcewake domains are needed to access a register

Parameters**struct intel_uncore *uncore**

pointer to struct intel_uncore

i915_reg_t reg

register in question

unsigned int op

operation bitmask of FW_REG_READ and/or FW_REG_WRITE

Description

Returns a set of forcewake domains required to be taken with for example `intel_uncore_forcewake_get` for the specified register to be accessible in the specified mode (read, write or read/write) with raw mmio accessors.

NOTE

On Gen6 and Gen7 write forcewake domain (FORCEWAKE_RENDER) requires the callers to do FIFO management on their own or risk losing writes.

Interrupt Handling

These functions provide the basic support for enabling and disabling the interrupt handling support. There's a lot more functionality in `i915_irq.c` and related files, but that will be described in separate chapters.

void **intel_irq_init**(struct drm_i915_private *dev_priv)
initializes irq support

Parameters

struct drm_i915_private *dev_priv
i915 device instance

Description

This function initializes all the irq support including work items, timers and all the vtables. It does not setup the interrupt itself though.

void **intel_runtime_pm_disable_interrupts**(struct drm_i915_private *dev_priv)
runtime interrupt disabling

Parameters

struct drm_i915_private *dev_priv
i915 device instance

Description

This function is used to disable interrupts at runtime, both in the runtime pm and the system suspend/resume code.

void **intel_runtime_pm_enable_interrupts**(struct drm_i915_private *dev_priv)
runtime interrupt enabling

Parameters

struct drm_i915_private *dev_priv
i915 device instance

Description

This function is used to enable interrupts at runtime, both in the runtime pm and the system suspend/resume code.

Intel GVT-g Guest Support(vGPU)

Intel GVT-g is a graphics virtualization technology which shares the GPU among multiple virtual machines on a time-sharing basis. Each virtual machine is presented a virtual GPU (vGPU), which has equivalent features as the underlying physical GPU (pGPU), so i915 driver can run seamlessly in a virtual machine. This file provides vGPU specific optimizations when running in a virtual machine, to reduce the complexity of vGPU emulation and to improve the overall performance.

A primary function introduced here is so-called “address space ballooning” technique. Intel GVT-g partitions global graphics memory among multiple VMs, so each VM can directly access a portion of the memory without hypervisor's intervention, e.g. filling textures or queuing commands. However with the partitioning an unmodified i915 driver would assume a smaller

graphics memory starting from address ZERO, then requires vGPU emulation module to translate the graphics address between ‘guest view’ and ‘host view’, for all registers and command opcodes which contain a graphics memory address. To reduce the complexity, Intel GVT-g introduces “address space ballooning”, by telling the exact partitioning knowledge to each guest i915 driver, which then reserves and prevents non-allocated portions from allocation. Thus vGPU emulation module only needs to scan and validate graphics addresses without complexity of address translation.

```
void intel_vgpu_detect(struct drm_i915_private *dev_priv)
    detect virtual GPU
```

Parameters

```
struct drm_i915_private *dev_priv
    i915 device private
```

Description

This function is called at the initialization stage, to detect whether running on a vGPU.

```
void intel_vgt_deballoon(struct i915_gggt *gggt)
    deballoon reserved graphics address trunks
```

Parameters

```
struct i915_gggt *gggt
    the global GGTT from which we reserved earlier
```

Description

This function is called to deallocate the ballooned-out graphic memory, when driver is unloaded or when ballooning fails.

```
int intel_vgt_balloon(struct i915_gggt *gggt)
    balloon out reserved graphics address trunks
```

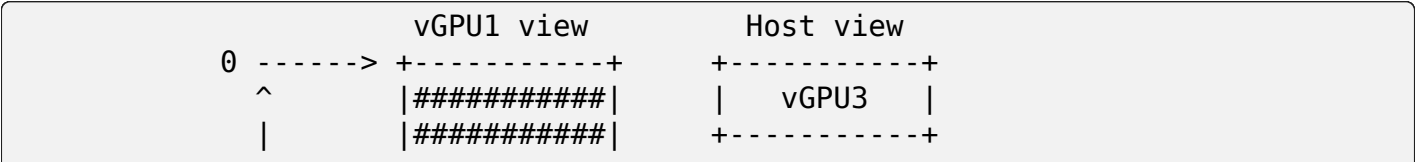
Parameters

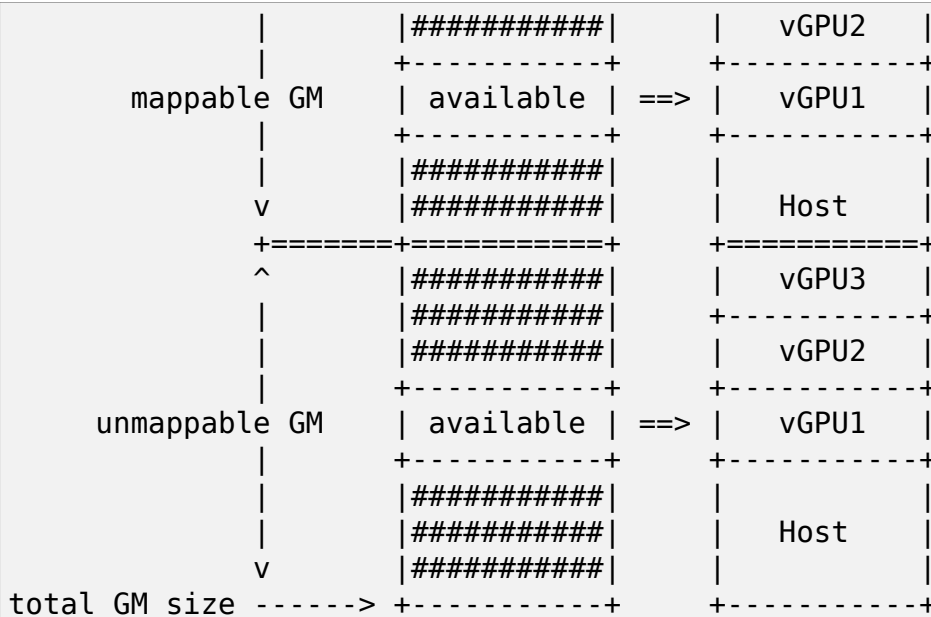
```
struct i915_gggt *gggt
    the global GGTT from which to reserve
```

Description

This function is called at the initialization stage, to balloon out the graphic address space allocated to other vGPUs, by marking these spaces as reserved. The ballooning related knowledge(starting address and size of the mappable/unmappable graphic memory) is described in the vgt_if structure in a reserved mmio range.

To give an example, the drawing below depicts one typical scenario after ballooning. Here the vGPU1 has 2 pieces of graphic address spaces ballooned out each for the mappable and the non-mappable part. From the vGPU1 point of view, the total size is the same as the physical one, with the start address of its graphic space being zero. Yet there are some portions ballooned out(the shadow part, which are marked as reserved by drm allocator). From the host point of view, the graphic address space is partitioned by multiple vGPUs in different VMs.



**Return**

zero on success, non-zero if configuration invalid or ballooning failed

Intel GVT-g Host Support(vGPU device model)

Intel GVT-g is a graphics virtualization technology which shares the GPU among multiple virtual machines on a time-sharing basis. Each virtual machine is presented a virtual GPU (vGPU), which has equivalent features as the underlying physical GPU (pGPU), so i915 driver can run seamlessly in a virtual machine.

To virtualize GPU resources GVT-g driver depends on hypervisor technology e.g KVM/VFIO/mdev, Xen, etc. to provide resource access trapping capability and be virtualized within GVT-g device module. More architectural design doc is available on <https://01.org/group/2230/documentation-list>.

int **intel_gvt_init**(struct drm_i915_private *dev_priv)
 initialize GVT components

Parameters

struct drm_i915_private *dev_priv
 drm i915 private data

Description

This function is called at the initialization stage to create a GVT device.

Return

Zero on success, negative error code if failed.

void **intel_gvt_driver_remove**(struct drm_i915_private *dev_priv)
 cleanup GVT components when i915 driver is unbinding

Parameters

```
struct drm_i915_private *dev_priv
    drm i915 private *
```

Description

This function is called at the i915 driver unloading stage, to shutdown GVT components and release the related resources.

```
void intel_gvt_resume(struct drm_i915_private *dev_priv)
    GVT resume routine wapper
```

Parameters

```
struct drm_i915_private *dev_priv
    drm i915 private *
```

Description

This function is called at the i915 driver resume stage to restore required HW status for GVT so that vGPU can continue running after resumed.

Workarounds

Hardware workarounds are register programming documented to be executed in the driver that fall outside of the normal programming sequences for a platform. There are some basic categories of workarounds, depending on how/when they are applied:

- Context workarounds: workarounds that touch registers that are saved/restored to/from the HW context image. The list is emitted (via Load Register Immediate commands) once when initializing the device and saved in the default context. That default context is then used on every context creation to have a “primed golden context”, i.e. a context image that already contains the changes needed to all the registers.

Context workarounds should be implemented in the `*_ctx_workarounds_init()` variants respective to the targeted platforms.

- Engine workarounds: the list of these WAs is applied whenever the specific engine is reset. It's also possible that a set of engine classes share a common power domain and they are reset together. This happens on some platforms with render and compute engines. In this case (at least) one of them need to keep the workaround programming: the approach taken in the driver is to tie those workarounds to the first compute/render engine that is registered. When executing with GuC submission, engine resets are outside of kernel driver control, hence the list of registers involved in written once, on engine initialization, and then passed to GuC, that saves/restores their values before/after the reset takes place. See `drivers/gpu/drm/i915/gt/uc/intel_guc_ads.c` for reference.

Workarounds for registers specific to RCS and CCS should be implemented in `rccs_engine_wa_init()` and `ccs_engine_wa_init()`, respectively; those for registers belonging to BCS, VCS or VECS should be implemented in `xcs_engine_wa_init()`. Workarounds for registers not belonging to a specific engine's MMIO range but that are part of the common RCS/CCS reset domain should be implemented in `general_render_compute_wa_init()`. The settings about the CCS load balancing should be added in `ccs_engine_wa_mode()`.

- GT workarounds: the list of these WAs is applied whenever these registers revert to their default values: on GPU reset, suspend/resume¹, etc.

¹ Technically, some registers are powercontext saved & restored, so they survive a suspend/resume. In practice,

GT workarounds should be implemented in the `*_gt_workarounds_init()` variants respective to the targeted platforms.

- Register whitelist: some workarounds need to be implemented in userspace, but need to touch privileged registers. The whitelist in the kernel instructs the hardware to allow the access to happen. From the kernel side, this is just a special case of a MMIO workaround (as we write the list of these to/be-whitelisted registers to some special HW registers).

Register whitelisting should be done in the `*_whitelist_build()` variants respective to the targeted platforms.

- Workaround batchbuffers: buffers that get executed automatically by the hardware on every HW context restore. These buffers are created and programmed in the default context so the hardware always go through those programming sequences when switching contexts. The support for workaround batchbuffers is enabled these hardware mechanisms:
 1. `INDIRECT_CTX`: A batchbuffer and an offset are provided in the default context, pointing the hardware to jump to that location when that offset is reached in the context restore. Workaround batchbuffer in the driver currently uses this mechanism for all platforms.
 2. `BB_PER_CTX_PTR`: A batchbuffer is provided in the default context, pointing the hardware to a buffer to continue executing after the engine registers are restored in a context restore sequence. This is currently not used in the driver.
- Other: There are WAs that, due to their nature, cannot be applied from a central place. Those are peppered around the rest of the code, as needed. Workarounds related to the display IP are the main example.

10.2.2 Display Hardware Handling

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

Mode Setting Infrastructure

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

Frontbuffer Tracking

Many features require us to track changes to the currently active frontbuffer, especially rendering targeted at the frontbuffer.

To be able to do so we track frontbuffers using a bitmask for all possible frontbuffer slots through `intel_frontbuffer_track()`. The functions in this file are then called when the contents of the frontbuffer are invalidated, when frontbuffer rendering has stopped again to flush out all the changes and when the frontbuffer is exchanged with a flip. Subsystems interested in frontbuffer changes (e.g. PSR, FBC, DRRS) should directly put their callbacks into the relevant places and filter for the frontbuffer slots that they are interested in.

writing them again is not too costly and simplifies things, so it's the approach taken in the driver.

On a high level there are two types of powersaving features. The first one work like a special cache (FBC and PSR) and are interested when they should stop caching and when to restart caching. This is done by placing callbacks into the invalidate and the flush functions: At invalidate the caching must be stopped and at flush time it can be restarted. And maybe they need to know when the frontbuffer changes (e.g. when the hw doesn't initiate an invalidate and flush on its own) which can be achieved with placing callbacks into the flip functions.

The other type of display power saving feature only cares about busyness (e.g. DRRS). In that case all three (invalidate, flush and flip) indicate busyness. There is no direct way to detect idleness. Instead an idle timer work delayed work should be started from the flush and flip functions and cancelled as soon as busyness is detected.

bool **intel_frontbuffer_invalidate**(struct intel_frontbuffer *front, enum fb_op_origin origin)

invalidate frontbuffer object

Parameters

struct intel_frontbuffer *front

GEM object to invalidate

enum fb_op_origin origin

which operation caused the invalidation

Description

This function gets called every time rendering on the given object starts and frontbuffer caching (fbc, low refresh rate for DRRS, panel self refresh) must be invalidated. For ORIGIN_CS any subsequent invalidation will be delayed until the rendering completes or a flip on this frontbuffer plane is scheduled.

void **intel_frontbuffer_flush**(struct intel_frontbuffer *front, enum fb_op_origin origin)

flush frontbuffer object

Parameters

struct intel_frontbuffer *front

GEM object to flush

enum fb_op_origin origin

which operation caused the flush

Description

This function gets called every time rendering on the given object has completed and frontbuffer caching can be started again.

void **frontbuffer_flush**(struct drm_i915_private *i915, unsigned int frontbuffer_bits, enum fb_op_origin origin)

flush frontbuffer

Parameters

struct drm_i915_private *i915

i915 device

unsigned int frontbuffer_bits

frontbuffer plane tracking bits

enum fb_op_origin origin

which operation caused the flush

Description

This function gets called every time rendering on the given planes has completed and frontbuffer caching can be started again. Flushes will get delayed if they're blocked by some outstanding asynchronous rendering.

Can be called without any locks held.

```
void intel_frontbuffer_flip_prepare(struct drm_i915_private *i915, unsigned  
                                frontbuffer_bits)
```

prepare asynchronous frontbuffer flip

Parameters

struct drm_i915_private *i915
i915 device

unsigned frontbuffer_bits
frontbuffer plane tracking bits

Description

This function gets called after scheduling a flip on **obj**. The actual frontbuffer flushing will be delayed until completion is signalled with `intel_frontbuffer_flip_complete`. If an invalidate happens in between this flush will be cancelled.

Can be called without any locks held.

```
void intel_frontbuffer_flip_complete(struct drm_i915_private *i915, unsigned  
                                    frontbuffer_bits)
```

complete asynchronous frontbuffer flip

Parameters

struct drm_i915_private *i915
i915 device

unsigned frontbuffer_bits
frontbuffer plane tracking bits

Description

This function gets called after the flip has been latched and will complete on the next vblank. It will execute the flush if it hasn't been cancelled yet.

Can be called without any locks held.

```
void intel_frontbuffer_flip(struct drm_i915_private *i915, unsigned frontbuffer_bits)  
    synchronous frontbuffer flip
```

Parameters

struct drm_i915_private *i915
i915 device

unsigned frontbuffer_bits
frontbuffer plane tracking bits

Description

This function gets called after scheduling a flip on **obj**. This is for synchronous plane updates which will happen on the next vblank and which will not get delayed by pending gpu rendering.

Can be called without any locks held.

```
void intel_frontbuffer_track(struct intel_frontbuffer *old, struct intel_frontbuffer *new,  
                           unsigned int frontbuffer_bits)  
    update frontbuffer tracking
```

Parameters

struct intel_frontbuffer *old
current buffer for the frontbuffer slots

struct intel_frontbuffer *new
new buffer for the frontbuffer slots

unsigned int frontbuffer_bits
bitmask of frontbuffer slots

Description

This updates the frontbuffer tracking bits **frontbuffer_bits** by clearing them from **old** and setting them in **new**. Both **old** and **new** can be NULL.

Display FIFO Underrun Reporting

The i915 driver checks for display fifo underruns using the interrupt signals provided by the hardware. This is enabled by default and fairly useful to debug display issues, especially watermark settings.

If an underrun is detected this is logged into dmesg. To avoid flooding logs and occupying the cpu underrun interrupts are disabled after the first occurrence until the next modeset on a given pipe.

Note that underrun detection on gmch platforms is a bit more ugly since there is no interrupt (despite that the signalling bit is in the PIPESTAT pipe interrupt register). Also on some other platforms underrun interrupts are shared, which means that if we detect an underrun we need to disable underrun reporting on all pipes.

The code also supports underrun detection on the PCH transcoder.

```
bool intel_set_cpu_fifo_underrun_reporting(struct drm_i915_private *dev_priv, enum  
                                           pipe pipe, bool enable)  
    set cpu fifo underrun reporting state
```

Parameters

struct drm_i915_private *dev_priv
i915 device instance

enum pipe pipe
(CPU) pipe to set state for

bool enable
whether underruns should be reported or not

Description

This function sets the fifo underrun state for **pipe**. It is used in the modeset code to avoid false positives since on many platforms underruns are expected when disabling or enabling the pipe.

Notice that on some platforms disabling underrun reports for one pipe disables for all due to shared interrupts. Actual reporting is still per-pipe though.

Returns the previous state of underrun reporting.

bool **intel_set_pch_fifo_underrun_reporting**(struct drm_i915_private *dev_priv, enum pipe pch_transcoder, bool enable)

set PCH fifo underrun reporting state

Parameters

struct drm_i915_private *dev_priv
i915 device instance

enum pipe pch_transcoder
the PCH transcoder (same as pipe on IVB and older)

bool enable
whether underruns should be reported or not

Description

This function makes us disable or enable PCH fifo underruns for a specific PCH transcoder. Notice that on some PCHs (e.g. CPT/PPT), disabling FIFO underrun reporting for one transcoder may also disable all the other PCH error interrupts for the other transcoders, due to the fact that there's just one interrupt mask/enable bit for all the transcoders.

Returns the previous state of underrun reporting.

void **intel_cpu_fifo_underrun_irq_handler**(struct drm_i915_private *dev_priv, enum pipe pipe)

handle CPU fifo underrun interrupt

Parameters

struct drm_i915_private *dev_priv
i915 device instance

enum pipe pipe
(CPU) pipe to set state for

Description

This handles a CPU fifo underrun interrupt, generating an underrun warning into dmesg if underrun reporting is enabled and then disables the underrun interrupt to avoid an irq storm.

void **intel_pch_fifo_underrun_irq_handler**(struct drm_i915_private *dev_priv, enum pipe pch_transcoder)

handle PCH fifo underrun interrupt

Parameters

struct drm_i915_private *dev_priv
i915 device instance

enum pipe pch_transcoder

the PCH transcoder (same as pipe on IVB and older)

Description

This handles a PCH fifo underrun interrupt, generating an underrun warning into dmesg if underrun reporting is enabled and then disables the underrun interrupt to avoid an irq storm.

void **intel_check_cpu_fifo_underruns**(struct drm_i915_private *dev_priv)

check for CPU fifo underruns immediately

Parameters

struct drm_i915_private *dev_priv

i915 device instance

Description

Check for CPU fifo underruns immediately. Useful on IVB/HSW where the shared error interrupt may have been disabled, and so CPU fifo underruns won't necessarily raise an interrupt, and on GMCH platforms where underruns never raise an interrupt.

void **intel_check_pch_fifo_underruns**(struct drm_i915_private *dev_priv)

check for PCH fifo underruns immediately

Parameters

struct drm_i915_private *dev_priv

i915 device instance

Description

Check for PCH fifo underruns immediately. Useful on CPT/PPT where the shared error interrupt may have been disabled, and so PCH fifo underruns won't necessarily raise an interrupt.

Plane Configuration

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

Atomic Plane Helpers

The functions here are used by the atomic plane helper functions to implement legacy plane updates (i.e., `drm_plane->update_plane()` and `drm_plane->disable_plane()`). This allows plane updates to use the atomic state infrastructure and perform plane updates as separate prepare/check/commit/cleanup steps.

struct *drm_plane_state* ***intel_plane_duplicate_state**(struct *drm_plane* *plane)

duplicate plane state

Parameters

struct drm_plane *plane

drm plane

Description

Allocates and returns a copy of the plane state (both common and Intel-specific) for the specified plane.

Return

The newly allocated plane state, or NULL on failure.

void **intel_plane_destroy_state**(struct *drm_plane* *plane, struct *drm_plane_state* *state)
destroy plane state

Parameters

struct drm_plane *plane
drm plane

struct drm_plane_state *state
state object to destroy

Description

Destroys the plane state (both common and Intel-specific) for the specified plane.

int **intel_prepare_plane_fb**(struct *drm_plane* *_plane, struct *drm_plane_state* *_new_plane_state)
Prepare fb for usage on plane

Parameters

struct drm_plane *_plane
drm plane to prepare for

struct drm_plane_state *_new_plane_state
the plane state being prepared

Description

Prepares a framebuffer for usage on a display plane. Generally this involves pinning the underlying object and updating the frontbuffer tracking bits. Some older platforms need special physical address handling for cursor planes.

Returns 0 on success, negative error code on failure.

void **intel_cleanup_plane_fb**(struct *drm_plane* *plane, struct *drm_plane_state* *_old_plane_state)
Cleans up an fb after plane use

Parameters

struct drm_plane *plane
drm plane to clean up for

struct drm_plane_state *_old_plane_state
the state from the previous modeset

Description

Cleans up a framebuffer that has just been removed from a plane.

Asynchronous Page Flip

Asynchronous page flip is the implementation for the `DRM_MODE_PAGE_FLIP_ASYNC` flag. Currently async flip is only supported via the `drmModePageFlip` IOCTL. Correspondingly, support is currently added for primary plane only.

Async flip can only change the plane surface address, so anything else changing is rejected from the `intel_async_flip_check_hw()` function. Once this check is cleared, flip done interrupt is enabled using the `intel_crtc_enable_flip_done()` function.

As soon as the surface address register is written, flip done interrupt is generated and the requested events are sent to the userspace in the interrupt handler itself. The timestamp and sequence sent during the flip done event correspond to the last vblank and have no relation to the actual time when the flip done event was sent.

Output Probing

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

Hotplug

Simply put, hotplug occurs when a display is connected to or disconnected from the system. However, there may be adapters and docking stations and Display Port short pulses and MST devices involved, complicating matters.

Hotplug in i915 is handled in many different levels of abstraction.

The platform dependent interrupt handling code in `i915_irq.c` enables, disables, and does preliminary handling of the interrupts. The interrupt handlers gather the hotplug detect (HPD) information from relevant registers into a platform independent mask of hotplug pins that have fired.

The platform independent interrupt handler `intel_hpd_irq_handler()` in `intel_hotplug.c` does hotplug irq storm detection and mitigation, and passes further processing to appropriate bottom halves (Display Port specific and regular hotplug).

The Display Port work function `i915_digport_work_func()` calls into `intel_dp_hpd_pulse()` via hooks, which handles DP short pulses and DP MST long pulses, with failures and non-MST long pulses triggering regular hotplug processing on the connector.

The regular hotplug work function `i915_hotplug_work_func()` calls connector detect hooks, and, if connector status changes, triggers sending of hotplug uevent to userspace via `drm_kms_helper_hotplug_event()`.

Finally, the userspace is responsible for triggering a modeset upon receiving the hotplug uevent, disabling or enabling the crtc as needed.

The hotplug interrupt storm detection and mitigation code keeps track of the number of interrupts per hotplug pin per a period of time, and if the number of interrupts exceeds a certain threshold, the interrupt is disabled for a while before being re-enabled. The intention is to mitigate issues raising from broken hardware triggering massive amounts of interrupts and grinding the system to a halt.

Current implementation expects that hotplug interrupt storm will not be seen when display port sink is connected, hence on platforms whose DP callback is handled by `i915_digport_work_func` reenabling of hpd is not performed (it was never expected to be disabled in the first place ;)) this is specific to DP sinks handled by this routine and any other display such as HDMI or DVI enabled on the same port will have proper logic since it will use `i915_hotplug_work_func` where this logic is handled.

enum hpd_pin **intel_hpd_pin_default**(struct drm_i915_private *dev_priv, enum *port* port)
return default pin associated with certain port.

Parameters

struct drm_i915_private *dev_priv
private driver data pointer

enum port port
the hpd port to get associated pin

Description

It is only valid and used by digital port encoder.

Return pin that is associated with **port**.

bool **intel_hpd_irq_storm_detect**(struct drm_i915_private *dev_priv, enum hpd_pin pin, bool long_hpd)
gather stats and detect HPD IRQ storm on a pin

Parameters

struct drm_i915_private *dev_priv
private driver data pointer

enum hpd_pin pin
the pin to gather stats on

bool long_hpd
whether the HPD IRQ was long or short

Description

Gather stats about HPD IRQs from the specified **pin**, and detect IRQ storms. Only the pin specific stats and state are changed, the caller is responsible for further action.

The number of IRQs that are allowed within **HPD_STORM_DETECT_PERIOD** is stored in **dev_priv->display.hotplug.hpd_storm_threshold** which defaults to **HPD_STORM_DEFAULT_THRESHOLD**. Long IRQs count as +10 to this threshold, and short IRQs count as +1. If this threshold is exceeded, it's considered an IRQ storm and the IRQ state is set to **HPD_MARK_DISABLED**.

By default, most systems will only count long IRQs towards **dev_priv->display.hotplug.hpd_storm_threshold**. However, some older systems also suffer from short IRQ storms and must also track these. Because short IRQ storms are naturally caused by sideband interactions with DP MST devices, short IRQ detection is only enabled for systems without DP MST support. Systems which are new enough to support DP MST are far less likely to suffer from IRQ storms at all, so this is fine.

The HPD threshold can be controlled through `i915_hpd_storm_ctl` in debugfs, and should only be adjusted for automated hotplug testing.

Return true if an IRQ storm was detected on **pin**.

void **intel_hpd_trigger_irq**(struct intel_digital_port *dig_port)
trigger an hpd irq event for a port

Parameters

struct intel_digital_port *dig_port
digital port

Description

Trigger an HPD interrupt event for the given port, emulating a short pulse generated by the sink, and schedule the dig port work to handle it.

void **intel_hpd_irq_handler**(struct drm_i915_private *dev_priv, u32 pin_mask, u32 long_mask)
main hotplug irq handler

Parameters

struct drm_i915_private *dev_priv
drm_i915_private

u32 pin_mask
a mask of hpd pins that have triggered the irq

u32 long_mask
a mask of hpd pins that may be long hpd pulses

Description

This is the main hotplug irq handler for all platforms. The platform specific irq handlers call the platform specific hotplug irq handlers, which read and decode the appropriate registers into bitmasks about hpd pins that have triggered (**pin_mask**), and which of those pins may be long pulses (**long_mask**). The **long_mask** is ignored if the port corresponding to the pin is not a digital port.

Here, we do hotplug irq storm detection and mitigation, and pass further processing to appropriate bottom halves.

void **intel_hpd_init**(struct drm_i915_private *dev_priv)
initializes and enables hpd support

Parameters

struct drm_i915_private *dev_priv
i915 device instance

Description

This function enables the hotplug support. It requires that interrupts have already been enabled with `intel_irq_init_hw()`. From this point on hotplug and poll request can run concurrently to other code, so locking rules must be obeyed.

This is a separate step from interrupt enabling to simplify the locking rules in the driver load and resume code.

Also see: [`intel_hpd_poll_enable\(\)`](#) and [`intel_hpd_poll_disable\(\)`](#).

void **intel_hpd_poll_enable**(struct drm_i915_private *dev_priv)
enable polling for connectors with hpd

Parameters

struct drm_i915_private *dev_priv
i915 device instance

Description

This function enables polling for all connectors which support HPD. Under certain conditions HPD may not be functional. On most Intel GPUs, this happens when we enter runtime suspend. On Valleyview and Cherryview systems, this also happens when we shut off all of the powerwells.

Since this function can get called in contexts where we're already holding dev->mode_config.mutex, we do the actual hotplug enabling in a separate worker.

Also see: [intel_hpd_init\(\)](#) and [intel_hpd_poll_disable\(\)](#).

void **intel_hpd_poll_disable**(struct drm_i915_private *dev_priv)
disable polling for connectors with hpd

Parameters

struct drm_i915_private *dev_priv
i915 device instance

Description

This function disables polling for all connectors which support HPD. Under certain conditions HPD may not be functional. On most Intel GPUs, this happens when we enter runtime suspend. On Valleyview and Cherryview systems, this also happens when we shut off all of the powerwells.

Since this function can get called in contexts where we're already holding dev->mode_config.mutex, we do the actual hotplug enabling in a separate worker.

Also used during driver init to initialize connector->polled appropriately for all connectors.

Also see: [intel_hpd_init\(\)](#) and [intel_hpd_poll_enable\(\)](#).

High Definition Audio

The graphics and audio drivers together support High Definition Audio over HDMI and Display Port. The audio programming sequences are divided into audio codec and controller enable and disable sequences. The graphics driver handles the audio codec sequences, while the audio driver handles the audio controller sequences.

The disable sequences must be performed before disabling the transcoder or port. The enable sequences may only be performed after enabling the transcoder and port, and after completed link training. Therefore the audio enable/disable sequences are part of the modeset sequence.

The codec and controller sequences could be done either parallel or serial, but generally the ELDV/PD change in the codec sequence indicates to the audio driver that the controller sequence should start. Indeed, most of the co-operation between the graphics and audio drivers is handled via audio related registers. (The notable exception is the power management, not covered here.)

The struct *intel_i915_audio_component* is used to interact between the graphics and audio drivers. The struct *intel_i915_audio_component_ops* **ops** in it is defined in graphics driver and called in audio driver. The struct *intel_i915_audio_component_audio_ops* **audio_ops** is called from i915 driver.

```
void intel_audio_codec_enable(struct intel_encoder *encoder, const struct intel_crtc_state
                             *crtc_state, const struct drm_connector_state *conn_state)
```

Enable the audio codec for HD audio

Parameters

struct intel_encoder *encoder
encoder on which to enable audio

const struct intel_crtc_state *crtc_state
pointer to the current crtc state.

const struct drm_connector_state *conn_state
pointer to the current connector state.

Description

The enable sequences may only be performed after enabling the transcoder and port, and after completed link training.

```
void intel_audio_codec_disable(struct intel_encoder *encoder, const struct intel_crtc_state
                              *old_crtc_state, const struct drm_connector_state
                              *old_conn_state)
```

Disable the audio codec for HD audio

Parameters

struct intel_encoder *encoder
encoder on which to disable audio

const struct intel_crtc_state *old_crtc_state
pointer to the old crtc state.

const struct drm_connector_state *old_conn_state
pointer to the old connector state.

Description

The disable sequences must be performed before disabling the transcoder or port.

```
void intel_audio_hooks_init(struct drm_i915_private *i915)
    Set up chip specific audio hooks
```

Parameters

struct drm_i915_private *i915
device private

```
void i915_audio_component_init(struct drm_i915_private *i915)
    initialize and register the audio component
```

Parameters

struct drm_i915_private *i915
i915 device instance

Description

This will register with the component framework a child component which will bind dynamically to the `snd_hda_intel` driver's corresponding master component when the latter is registered. During binding the child initializes an instance of `struct i915_audio_component` which it receives from the master. The master can then start to use the interface defined by this struct. Each side can break the binding at any point by deregistering its own component after which each side's component unbind callback is called.

We ignore any error during registration and continue with reduced functionality (i.e. without HDMI audio).

void **i915_audio_component_cleanup**(struct drm_i915_private *i915)

deregister the audio component

Parameters

struct drm_i915_private *i915

i915 device instance

Description

Deregisters the audio component, breaking any existing binding to the corresponding `snd_hda_intel` driver's master component.

void **intel_audio_init**(struct drm_i915_private *i915)

Initialize the audio driver either using component framework or using lpe audio bridge

Parameters

struct drm_i915_private *i915

the i915 drm device private data

void **intel_audio_deinit**(struct drm_i915_private *i915)

deinitialize the audio driver

Parameters

struct drm_i915_private *i915

the i915 drm device private data

struct **i915_audio_component**

Used for direct communication between i915 and hda drivers

Definition:

```
struct i915_audio_component {
    struct drm_audio_component    base;
    int aud_sample_rate[MAX_PORTS];
};
```

Members

base

the `drm_audio_component` base class

aud_sample_rate

the array of audio sample rate per port

Intel HDMI LPE Audio Support

Motivation: Atom platforms (e.g. valleyview and cherryTrail) integrates a DMA-based interface as an alternative to the traditional HDAudio path. While this mode is unrelated to the LPE aka SST audio engine, the documentation refers to this mode as LPE so we keep this notation for the sake of consistency.

The interface is handled by a separate standalone driver maintained in the ALSA subsystem for simplicity. To minimize the interaction between the two subsystems, a bridge is setup between the hdmi-lpe-audio and i915: 1. Create a platform device to share MMIO/IRQ resources 2. Make the platform device child of i915 device for runtime PM. 3. Create IRQ chip to forward the LPE audio irqs. the hdmi-lpe-audio driver probes the lpe audio device and creates a new sound card

Threats: Due to the restriction in Linux platform device model, user need manually uninstall the hdmi-lpe-audio driver before uninstalling i915 module, otherwise we might run into use-after-free issues after i915 removes the platform device: even though hdmi-lpe-audio driver is released, the modules is still in “installed” status.

Implementation: The MMIO/REG platform resources are created according to the registers specification. When forwarding LPE audio irqs, the flow control handler selection depends on the platform, for example on valleyview handle_simple_irq is enough.

void **intel_lpe_audio_irq_handler**(struct drm_i915_private *dev_priv)
forwards the LPE audio irq

Parameters

struct drm_i915_private *dev_priv
the i915 drm device private data

Description

the LPE Audio irq is forwarded to the irq handler registered by LPE audio driver.

int **intel_lpe_audio_init**(struct drm_i915_private *dev_priv)
detect and setup the bridge between HDMI LPE Audio driver and i915

Parameters

struct drm_i915_private *dev_priv
the i915 drm device private data

Return

0 if successful. non-zero if detection or llocation/initialization fails

void **intel_lpe_audio_tearardown**(struct drm_i915_private *dev_priv)
destroy the bridge between HDMI LPE audio driver and i915

Parameters

struct drm_i915_private *dev_priv
the i915 drm device private data

Description

release all the resources for LPE audio <-> i915 bridge.

```
void intel_lpe_audio_notify(struct drm_i915_private *dev_priv, enum transcoder
                           cpu_transcoder, enum port port, const void *eld, int ls_clock,
                           bool dp_output)
    notify lpe audio event audio driver and i915
```

Parameters

struct drm_i915_private *dev_priv
the i915 drm device private data

enum transcoder cpu_transcoder
CPU transcoder

enum port port
port

const void *eld
ELD data

int ls_clock
Link symbol clock in kHz

bool dp_output
Driving a DP output?

Description

Notify lpe audio driver of eld change.

Panel Self Refresh PSR (PSR/SRD)

Since Haswell Display controller supports Panel Self-Refresh on display panels which have a remote frame buffer (RFB) implemented according to PSR spec in eDP1.3. PSR feature allows the display to go to lower standby states when system is idle but display is on as it eliminates display refresh request to DDR memory completely as long as the frame buffer for that display is unchanged.

Panel Self Refresh must be supported by both Hardware (source) and Panel (sink).

PSR saves power by caching the framebuffer in the panel RFB, which allows us to power down the link and memory controller. For DSI panels the same idea is called “manual mode”.

The implementation uses the hardware-based PSR support which automatically enters/exits self-refresh mode. The hardware takes care of sending the required DP aux message and could even retrain the link (that part isn't enabled yet though). The hardware also keeps track of any framebuffer changes to know when to exit self-refresh mode again. Unfortunately that part doesn't work too well, hence why the i915 PSR support uses the software framebuffer tracking to make sure it doesn't miss a screen update. For this integration [*intel_psr_invalidate\(\)*](#) and [*intel_psr_flush\(\)*](#) get called by the framebuffer tracking code. Note that because of locking issues the self-refresh re-enable code is done from a work queue, which must be correctly synchronized/cancelled when shutting down the pipe.”

DC3CO (DC3 clock off)

On top of PSR2, GEN12 adds a intermediate power savings state that turns clock off automatically during PSR2 idle state. The smaller overhead of DC3co entry/exit vs. the overhead of

PSR2 deep sleep entry/exit allows the HW to enter a low-power state even when page flipping periodically (for instance a 30fps video playback scenario).

Every time a flips occurs PSR2 will get out of deep sleep state(if it was), so DC3CO is enabled and `tgl_dc3co_disable_work` is schedule to run after 6 frames, if no other flip occurs and the function above is executed, DC3CO is disabled and PSR2 is configured to enter deep sleep, resetting again in case of another flip. Front buffer modifications do not trigger DC3CO activation on purpose as it would bring a lot of complexity and most of the moderns systems will only use page flips.

```
void intel_psr_disable(struct intel_dp *intel_dp, const struct intel_crtc_state
                      *old_crtc_state)
```

Disable PSR

Parameters

```
struct intel_dp *intel_dp
```

Intel DP

```
const struct intel_crtc_state *old_crtc_state
```

old CRTC state

Description

This function needs to be called before disabling pipe.

```
void intel_psr_pause(struct intel_dp *intel_dp)
```

Pause PSR

Parameters

```
struct intel_dp *intel_dp
```

Intel DP

Description

This function need to be called after enabling psr.

```
void intel_psr_resume(struct intel_dp *intel_dp)
```

Resume PSR

Parameters

```
struct intel_dp *intel_dp
```

Intel DP

Description

This function need to be called after pausing psr.

```
void intel_psr_wait_for_idle_locked(const struct intel_crtc_state *new_crtc_state)
```

wait for PSR be ready for a pipe update

Parameters

```
const struct intel_crtc_state *new_crtc_state
```

new CRTC state

Description

This function is expected to be called from `pipe_update_start()` where it is not expected to race with PSR enable or disable.

void **intel_psr_invalidate**(struct drm_i915_private *dev_priv, unsigned frontbuffer_bits,
enum fb_op_origin origin)

Invalidate PSR

Parameters

struct drm_i915_private *dev_priv
i915 device

unsigned frontbuffer_bits
frontbuffer plane tracking bits

enum fb_op_origin origin
which operation caused the invalidate

Description

Since the hardware frontbuffer tracking has gaps we need to integrate with the software frontbuffer tracking. This function gets called every time frontbuffer rendering starts and a buffer gets dirtied. PSR must be disabled if the frontbuffer mask contains a buffer relevant to PSR.

Dirty frontbuffers relevant to PSR are tracked in `busy_frontbuffer_bits`.”

void **intel_psr_flush**(struct drm_i915_private *dev_priv, unsigned frontbuffer_bits, enum
fb_op_origin origin)

Flush PSR

Parameters

struct drm_i915_private *dev_priv
i915 device

unsigned frontbuffer_bits
frontbuffer plane tracking bits

enum fb_op_origin origin
which operation caused the flush

Description

Since the hardware frontbuffer tracking has gaps we need to integrate with the software frontbuffer tracking. This function gets called every time frontbuffer rendering has completed and flushed out to memory. PSR can be enabled again if no other frontbuffer relevant to PSR is dirty.

Dirty frontbuffers relevant to PSR are tracked in `busy_frontbuffer_bits`.

void **intel_psr_init**(struct *intel_dp* *intel_dp)
Init basic PSR work and mutex.

Parameters

struct intel_dp *intel_dp
Intel DP

Description

This function is called after the initializing connector. (the initializing of connector treats the handling of connector capabilities) And it initializes basic PSR stuff for each DP Encoder.

void **intel_psr_lock**(const struct intel_crtc_state *crtc_state)
grab PSR lock

Parameters

const struct intel_crtc_state *crtc_state
the crtc state

Description

This is initially meant to be used by around CRTC update, when vblank sensitive registers are updated and we need grab the lock before it to avoid vblank evasion.

void **intel_psr_unlock**(const struct intel_crtc_state *crtc_state)
release PSR lock

Parameters

const struct intel_crtc_state *crtc_state
the crtc state

Description

Release the PSR lock that was held during pipe update.

Frame Buffer Compression (FBC)

FBC tries to save memory bandwidth (and so power consumption) by compressing the amount of memory used by the display. It is total transparent to user space and completely handled in the kernel.

The benefits of FBC are mostly visible with solid backgrounds and variation-less patterns. It comes from keeping the memory footprint small and having fewer memory pages opened and accessed for refreshing the display.

i915 is responsible to reserve stolen memory for FBC and configure its offset on proper registers. The hardware takes care of all compress/decompress. However there are many known cases where we have to forcibly disable it to allow proper screen updates.

void **intel_fbc_disable**(struct intel_crtc *crtc)
disable FBC if it's associated with crtc

Parameters

struct intel_crtc *crtc
the CRTC

Description

This function disables FBC if it's associated with the provided CRTC.

void **intel_fbc_handle_fifo_underrun_irq**(struct drm_i915_private *i915)
disable FBC when we get a FIFO underrun

Parameters

struct drm_i915_private *i915
i915 device

Description

Without FBC, most underruns are harmless and don't really cause too many problems, except for an annoying message on dmesg. With FBC, underruns can become black screens or even worse, especially when paired with bad watermarks. So in order for us to be on the safe side, completely disable FBC in case we ever detect a FIFO underrun on any pipe. An underrun on any pipe already suggests that watermarks may be bad, so try to be as safe as possible.

This function is called from the IRQ handler.

```
void intel_fbc_init(struct drm_i915_private *i915)
    Initialize FBC
```

Parameters

```
struct drm_i915_private *i915
    the i915 device
```

Description

This function might be called during PM init process.

```
void intel_fbc_sanitize(struct drm_i915_private *i915)
    Sanitize FBC
```

Parameters

```
struct drm_i915_private *i915
    the i915 device
```

Description

Make sure FBC is initially disabled since we have no idea eg. into which parts of stolen it might be scribbling into.

Display Refresh Rate Switching (DRRS)

Display Refresh Rate Switching (DRRS) is a power conservation feature which enables switching between low and high refresh rates, dynamically, based on the usage scenario. This feature is applicable for internal panels.

Indication that the panel supports DRRS is given by the panel EDID, which would list multiple refresh rates for one resolution.

DRRS is of 2 types - static and seamless. Static DRRS involves changing refresh rate (RR) by doing a full modeset (may appear as a blink on screen) and is used in dock-undock scenario. Seamless DRRS involves changing RR without any visual effect to the user and can be used during normal system usage. This is done by programming certain registers.

Support for static/seamless DRRS may be indicated in the VBT based on inputs from the panel spec.

DRRS saves power by switching to low RR based on usage scenarios.

The implementation is based on frontbuffer tracking implementation. When there is a disturbance on the screen triggered by user activity or a periodic system activity, DRRS is disabled (RR is changed to high RR). When there is no movement on screen, after a timeout of 1 second, a switch to low RR is made.

For integration with frontbuffer tracking code, `intel_drrs_invalidate()` and `intel_drrs_flush()` are called.

DRRS can be further extended to support other internal panels and also the scenario of video playback wherein RR is set based on the rate requested by userspace.

void **intel_drrs_activate**(const struct intel_crtc_state *crtc_state)
activate DRRS

Parameters

const struct intel_crtc_state *crtc_state
the crtc state

Description

Activates DRRS on the crtc.

void **intel_drrs_deactivate**(const struct intel_crtc_state *old_crtc_state)
deactivate DRRS

Parameters

const struct intel_crtc_state *old_crtc_state
the old crtc state

Description

Deactivates DRRS on the crtc.

void **intel_drrs_invalidate**(struct drm_i915_private *dev_priv, unsigned int
frontbuffer_bits)
Disable Idleness DRRS

Parameters

struct drm_i915_private *dev_priv
i915 device

unsigned int frontbuffer_bits
frontbuffer plane tracking bits

Description

This function gets called everytime rendering on the given planes start. Hence DRRS needs to be Upclocked, i.e. (LOW_RR -> HIGH_RR).

Dirty frontbuffers relevant to DRRS are tracked in busy_frontbuffer_bits.

void **intel_drrs_flush**(struct drm_i915_private *dev_priv, unsigned int frontbuffer_bits)
Restart Idleness DRRS

Parameters

struct drm_i915_private *dev_priv
i915 device

unsigned int frontbuffer_bits
frontbuffer plane tracking bits

Description

This function gets called every time rendering on the given planes has completed or flip on a crtc is completed. So DRRS should be upclocked (LOW_RR -> HIGH_RR). And also Idleness detection should be started again, if no other planes are dirty.

Dirty frontbuffers relevant to DRRS are tracked in `busy_frontbuffer_bits`.

```
void intel_drrs_crtc_init(struct intel_crtc *crtc)
    Init DRRS for CRTC
```

Parameters

```
struct intel_crtc *crtc
    crtc
```

Description

This function is called only once at driver load to initialize basic DRRS stuff.

DPIO

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

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

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

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

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

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

For dual channel PHY (VLV/CHV):

```
pipe A == CMN/PLL/REF CH0
pipe B == CMN/PLL/REF CH1
port B == PCS/TX CH0
port C == PCS/TX CH1
```


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

For single channel PHY (CHV):

```
pipe C == CMN/PLL/REF CH0
port D == PCS/TX CH0
```

On BXT the entire PHY channel corresponds to the port. That means the PLL is also now associated with the port rather than the pipe, and so the clock needs to be routed to the appropriate transcoder. Port A PLL is directly connected to transcoder EDP and port B/C PLLs can be routed to any transcoder A/B/C.

Note: DDI0 is digital port B, DD1 is digital port C, and DDI2 is digital port D (CHV) or port A (BXT).

Dual channel PHY (VLV/CHV/BXT)

CH0		CH1		Display PHY
CMN/PLL/REF		CMN/PLL/REF		
-----		-----		
PCS01	PCS23	PCS01	PCS23	
-----	-----	-----	-----	
TX0	TX1	TX2	TX3	
TX0	TX1	TX2	TX3	

DDI0		DDI1		DP/HDMI ports

Single channel PHY (CHV/BXT)

CH0		Display PHY	
CMN/PLL/REF			

PCS01	PCS23		
-----	-----		
TX0	TX1	TX2	TX3
TX0	TX1	TX2	TX3

DDI2		DP/HDMI port	

DMC Firmware Support

From gen9 onwards we have newly added DMC (Display microcontroller) in display engine to save and restore the state of display engine when it enter into low-power state and comes back to normal.

```
void intel_dmc_load_program(struct drm_i915_private *i915)
    write the firmware from memory to register.
```

Parameters

```
struct drm_i915_private *i915
    i915 drm device.
```

Description

DMC firmware is read from a .bin file and kept in internal memory one time. Everytime display comes back from low power state this function is called to copy the firmware from internal memory to registers.

```
void intel_dmc_disable_program(struct drm_i915_private *i915)
    disable the firmware
```

Parameters

```
struct drm_i915_private *i915
    i915 drm device
```

Description

Disable all event handlers in the firmware, making sure the firmware is inactive after the display is uninitialized.

```
void intel_dmc_init(struct drm_i915_private *i915)
    initialize the firmware loading.
```

Parameters

```
struct drm_i915_private *i915
    i915 drm device.
```

Description

This function is called at the time of loading the display driver to read firmware from a .bin file and copied into a internal memory.

```
void intel_dmc_suspend(struct drm_i915_private *i915)
    prepare DMC firmware before system suspend
```

Parameters

```
struct drm_i915_private *i915
    i915 drm device
```

Description

Prepare the DMC firmware before entering system suspend. This includes flushing pending work items and releasing any resources acquired during init.

```
void intel_dmc_resume(struct drm_i915_private *i915)
    init DMC firmware during system resume
```

Parameters

```
struct drm_i915_private *i915
    i915 drm device
```

Description

Reinitialize the DMC firmware during system resume, reacquiring any resources released in [*intel_dmc_suspend\(\)*](#).

```
void intel_dmc_fini(struct drm_i915_private *i915)
    unload the DMC firmware.
```

Parameters

```
struct drm_i915_private *i915  
    i915 drm device.
```

Description

Firmware unloading includes freeing the internal memory and reset the firmware loading status.

Video BIOS Table (VBT)

The Video BIOS Table, or VBT, provides platform and board specific configuration information to the driver that is not discoverable or available through other means. The configuration is mostly related to display hardware. The VBT is available via the ACPI OpRegion or, on older systems, in the PCI ROM.

The VBT consists of a VBT Header (defined as *struct vbt_header*), a BDB Header (*struct bdb_header*), and a number of BIOS Data Blocks (BDB) that contain the actual configuration information. The VBT Header, and thus the VBT, begins with “\$VBT” signature. The VBT Header contains the offset of the BDB Header. The data blocks are concatenated after the BDB Header. The data blocks have a 1-byte Block ID, 2-byte Block Size, and Block Size bytes of data. (Block 53, the MIPI Sequence Block is an exception.)

The driver parses the VBT during load. The relevant information is stored in driver private data for ease of use, and the actual VBT is not read after that.

```
bool intel_bios_is_valid_vbt(const void *buf, size_t size)  
    does the given buffer contain a valid VBT
```

Parameters

```
const void *buf  
    pointer to a buffer to validate
```

```
size_t size  
    size of the buffer
```

Description

Returns true on valid VBT.

```
void intel_bios_init(struct drm_i915_private *i915)  
    find VBT and initialize settings from the BIOS
```

Parameters

```
struct drm_i915_private *i915  
    i915 device instance
```

Description

Parse and initialize settings from the Video BIOS Tables (VBT). If the VBT was not found in ACPI OpRegion, try to find it in PCI ROM first. Also initialize some defaults if the VBT is not present at all.

```
void intel_bios_driver_remove(struct drm_i915_private *i915)  
    Free any resources allocated by intel_bios_init()
```

Parameters

struct drm_i915_private *i915

i915 device instance

bool **intel_bios_is_tv_present**(struct drm_i915_private *i915)

is integrated TV present in VBT

Parameters

struct drm_i915_private *i915

i915 device instance

Description

Return true if TV is present. If no child devices were parsed from VBT, assume TV is present.

bool **intel_bios_is_lvds_present**(struct drm_i915_private *i915, u8 *i2c_pin)

is LVDS present in VBT

Parameters

struct drm_i915_private *i915

i915 device instance

u8 *i2c_pin

i2c pin for LVDS if present

Description

Return true if LVDS is present. If no child devices were parsed from VBT, assume LVDS is present.

bool **intel_bios_is_port_present**(struct drm_i915_private *i915, enum *port* port)

is the specified digital port present

Parameters

struct drm_i915_private *i915

i915 device instance

enum port port

port to check

Description

Return true if the device in port is present.

bool **intel_bios_is_dsi_present**(struct drm_i915_private *i915, enum *port* *port)

is DSI present in VBT

Parameters

struct drm_i915_private *i915

i915 device instance

enum port *port

port for DSI if present

Description

Return true if DSI is present, and return the port in port.

struct vbt_header

VBT Header structure

Definition:

```
struct vbt_header {
    u8 signature[20];
    u16 version;
    u16 header_size;
    u16 vbt_size;
    u8 vbt_checksum;
    u8 reserved0;
    u32 bdb_offset;
    u32 aim_offset[4];
};
```

Members**signature**

VBT signature, always starts with "\$VBT"

version

Version of this structure

header_size

Size of this structure

vbt_size

Size of VBT (VBT Header, BDB Header and data blocks)

vbt_checksum

Checksum

reserved0

Reserved

bdb_offsetOffset of *struct bdb_header* from beginning of VBT**aim_offset**

Offsets of add-in data blocks from beginning of VBT

struct bdb_header

BDB Header structure

Definition:

```
struct bdb_header {
    u8 signature[16];
    u16 version;
    u16 header_size;
    u16 bdb_size;
};
```

Members

signature

BDB signature "BIOS_DATA_BLOCK"

version

Version of the data block definitions

header_size

Size of this structure

bdb_size

Size of BDB (BDB Header and data blocks)

Display clocks

The display engine uses several different clocks to do its work. There are two main clocks involved that aren't directly related to the actual pixel clock or any symbol/bit clock of the actual output port. These are the core display clock (CDCLK) and RAWCLK.

CDCLK clocks most of the display pipe logic, and thus its frequency must be high enough to support the rate at which pixels are flowing through the pipes. Downscaling must also be accounted as that increases the effective pixel rate.

On several platforms the CDCLK frequency can be changed dynamically to minimize power consumption for a given display configuration. Typically changes to the CDCLK frequency require all the display pipes to be shut down while the frequency is being changed.

On SKL+ the DMC will toggle the CDCLK off/on during DC5/6 entry/exit. DMC will not change the active CDCLK frequency however, so that part will still be performed by the driver directly.

RAWCLK is a fixed frequency clock, often used by various auxiliary blocks such as AUX CH or backlight PWM. Hence the only thing we really need to know about RAWCLK is its frequency so that various dividers can be programmed correctly.

```
void intel_cdclk_init_hw(struct drm_i915_private *i915)
```

Initialize CDCLK hardware

Parameters

```
struct drm_i915_private *i915
```

i915 device

Description

Initialize CDCLK. This consists mainly of initializing `dev_priv->display.cdclk.hw` and sanitizing the state of the hardware if needed. This is generally done only during the display core initialization sequence, after which the DMC will take care of turning CDCLK off/on as needed.

```
void intel_cdclk_uninit_hw(struct drm_i915_private *i915)
```

Uninitialize CDCLK hardware

Parameters

```
struct drm_i915_private *i915
```

i915 device

Description

Uninitialize CDCLK. This is done only during the display core uninitialization sequence.

```
bool intel_cdclk_needs_modeset(const struct intel_cdclk_config *a, const struct
                               intel_cdclk_config *b)
```

Determine if changong between the CDCLK configurations requires a modeset on all pipes

Parameters

```
const struct intel_cdclk_config *a
    first CDCLK configuration
```

```
const struct intel_cdclk_config *b
    second CDCLK configuration
```

Return

True if changing between the two CDCLK configurations requires all pipes to be off, false if not.

```
bool intel_cdclk_can_cd2x_update(struct drm_i915_private *dev_priv, const struct
                                  intel_cdclk_config *a, const struct intel_cdclk_config *b)
```

Determine if changing between the two CDCLK configurations requires only a cd2x divider update

Parameters

```
struct drm_i915_private *dev_priv
    i915 device
```

```
const struct intel_cdclk_config *a
    first CDCLK configuration
```

```
const struct intel_cdclk_config *b
    second CDCLK configuration
```

Return

True if changing between the two CDCLK configurations can be done with just a cd2x divider update, false if not.

```
bool intel_cdclk_changed(const struct intel_cdclk_config *a, const struct intel_cdclk_config
                           *b)
```

Determine if two CDCLK configurations are different

Parameters

```
const struct intel_cdclk_config *a
    first CDCLK configuration
```

```
const struct intel_cdclk_config *b
    second CDCLK configuration
```

Return

True if the CDCLK configurations don't match, false if they do.

```
void intel_set_cdclk(struct drm_i915_private *dev_priv, const struct intel_cdclk_config
                      *cdclk_config, enum pipe pipe)
```

Push the CDCLK configuration to the hardware

Parameters

```
struct drm_i915_private *dev_priv
    i915 device
```

const struct intel_cdclk_config *cdclk_config
new CDCLK configuration

enum pipe pipe
pipe with which to synchronize the update

Description

Program the hardware based on the passed in CDCLK state, if necessary.

void **intel_set_cdclk_pre_plane_update**(struct intel_atomic_state *state)
Push the CDCLK state to the hardware

Parameters

struct intel_atomic_state *state
intel atomic state

Description

Program the hardware before updating the HW plane state based on the new CDCLK state, if necessary.

void **intel_set_cdclk_post_plane_update**(struct intel_atomic_state *state)
Push the CDCLK state to the hardware

Parameters

struct intel_atomic_state *state
intel atomic state

Description

Program the hardware after updating the HW plane state based on the new CDCLK state, if necessary.

void **intel_update_max_cdclk**(struct drm_i915_private *dev_priv)
Determine the maximum support CDCLK frequency

Parameters

struct drm_i915_private *dev_priv
i915 device

Description

Determine the maximum CDCLK frequency the platform supports, and also derive the maximum dot clock frequency the maximum CDCLK frequency allows.

void **intel_update_cdclk**(struct drm_i915_private *dev_priv)
Determine the current CDCLK frequency

Parameters

struct drm_i915_private *dev_priv
i915 device

Description

Determine the current CDCLK frequency.

u32 **intel_read_rawclk**(struct drm_i915_private *dev_priv)

Determine the current RAWCLK frequency

Parameters

struct drm_i915_private *dev_priv
i915 device

Description

Determine the current RAWCLK frequency. RAWCLK is a fixed frequency clock so this needs to be done only once.

void **intel_init_cdclk_hooks**(struct drm_i915_private *dev_priv)

Initialize CDCLK related modesetting hooks

Parameters

struct drm_i915_private *dev_priv
i915 device

Display PLLs

Display PLLs used for driving outputs vary by platform. While some have per-pipe or per-encoder dedicated PLLs, others allow the use of any PLL from a pool. In the latter scenario, it is possible that multiple pipes share a PLL if their configurations match.

This file provides an abstraction over display PLLs. The function [*intel_shared_dpll_init\(\)*](#) initializes the PLLs for the given platform. The users of a PLL are tracked and that tracking is integrated with the atomic modset interface. During an atomic operation, required PLLs can be reserved for a given CRTC and encoder configuration by calling [*intel_reserve_shared_dplls\(\)*](#) and previously reserved PLLs can be released with [*intel_release_shared_dplls\(\)*](#). Changes to the users are first staged in the atomic state, and then made effective by calling [*intel_shared_dpll_swap_state\(\)*](#) during the atomic commit phase.

struct [*intel_shared_dpll*](#) ***intel_get_shared_dpll_by_id**(struct drm_i915_private *dev_priv,
enum [*intel_dpll_id*](#) id)

get a DPLL given its id

Parameters

struct drm_i915_private *dev_priv
i915 device instance

enum intel_dpll_id id
pll id

Return

A pointer to the DPLL with **id**

void **intel_enable_shared_dpll**(const struct intel_crtc_state *crtc_state)

enable a CRTC's shared DPLL

Parameters

const struct intel_crtc_state *crtc_state
CRTC, and its state, which has a shared DPLL

Description

Enable the shared DPLL used by **crtc**.

void **intel_disable_shared_dpll**(const struct intel_crtc_state *crtc_state)
disable a CRTC's shared DPLL

Parameters

const struct intel_crtc_state *crtc_state
CRTC, and its state, which has a shared DPLL

Description

Disable the shared DPLL used by **crtc**.

void **intel_reference_shared_dpll_crtc**(const struct intel_crtc *crtc, const struct
intel_shared_dpll *pll, struct
intel_shared_dpll_state *shared_dpll_state)

Get a DPLL reference for a CRTC

Parameters

const struct intel_crtc *crtc
CRTC on which behalf the reference is taken

const struct intel_shared_dpll *pll
DPLL for which the reference is taken

struct intel_shared_dpll_state *shared_dpll_state
the DPLL atomic state in which the reference is tracked

Description

Take a reference for **pll** tracking the use of it by **crtc**.

void **intel_unreference_shared_dpll_crtc**(const struct intel_crtc *crtc, const struct
intel_shared_dpll *pll, struct
intel_shared_dpll_state *shared_dpll_state)

Drop a DPLL reference for a CRTC

Parameters

const struct intel_crtc *crtc
CRTC on which behalf the reference is dropped

const struct intel_shared_dpll *pll
DPLL for which the reference is dropped

struct intel_shared_dpll_state *shared_dpll_state
the DPLL atomic state in which the reference is tracked

Description

Drop a reference for **pll** tracking the end of use of it by **crtc**.

void **intel_shared_dpll_swap_state**(struct intel_atomic_state *state)
make atomic DPLL configuration effective

Parameters

struct intel_atomic_state *state
atomic state

Description

This is the dpll version of [drm_atomic_helper_swap_state\(\)](#) since the helper does not handle driver-specific global state.

For consistency with atomic helpers this function does a complete swap, i.e. it also puts the current state into **state**, even though there is no need for that at this moment.

void **icl_set_active_port_dpll**(struct intel_crtc_state *crtc_state, enum icl_port_dpll_id port_dpll_id)
select the active port DPLL for a given CRTC

Parameters

struct intel_crtc_state *crtc_state
state for the CRTC to select the DPLL for

enum icl_port_dpll_id port_dpll_id
the active **port_dpll_id** to select

Description

Select the given **port_dpll_id** instance from the DPLLs reserved for the CRTC.

void **intel_shared_dpll_init**(struct drm_i915_private *dev_priv)
Initialize shared DPLLs

Parameters

struct drm_i915_private *dev_priv
i915 device

Description

Initialize shared DPLLs for **dev_priv**.

int **intel_compute_shared_dplls**(struct intel_atomic_state *state, struct intel_crtc *crtc, struct intel_encoder *encoder)
compute DPLL state CRTC and encoder combination

Parameters

struct intel_atomic_state *state
atomic state

struct intel_crtc *crtc
CRTC to compute DPLLs for

struct intel_encoder *encoder
encoder

Description

This function computes the DPLL state for the given CRTC and encoder.

The new configuration in the atomic commit **state** is made effective by calling [*intel_shared_dpll_swap_state\(\)*](#).

Return

0 on success, negative error code on failure.

int **intel_reserve_shared_dpll**(struct intel_atomic_state *state, struct intel_crtc *crtc,
struct intel_encoder *encoder)

reserve DPLLs for CRTC and encoder combination

Parameters

struct intel_atomic_state *state

atomic state

struct intel_crtc *crtc

CRTC to reserve DPLLs for

struct intel_encoder *encoder

encoder

Description

This function reserves all required DPLLs for the given CRTC and encoder combination in the current atomic commit **state** and the new **crtc** atomic state.

The new configuration in the atomic commit **state** is made effective by calling [*intel_shared_dpll_swap_state\(\)*](#).

The reserved DPLLs should be released by calling [*intel_release_shared_dpll*](#)s()).

Return

0 if all required DPLLs were successfully reserved, negative error code otherwise.

void **intel_release_shared_dpll**s)(struct intel_atomic_state *state, struct intel_crtc *crtc)

end use of DPLLs by CRTC in atomic state

Parameters

struct intel_atomic_state *state

atomic state

struct intel_crtc *crtc

crtc from which the DPLLs are to be released

Description

This function releases all DPLLs reserved by [*intel_reserve_shared_dpll*](#)s()) from the current atomic commit **state** and the old **crtc** atomic state.

The new configuration in the atomic commit **state** is made effective by calling [*intel_shared_dpll_swap_state\(\)*](#).

void **intel_update_active_dpll**(struct intel_atomic_state *state, struct intel_crtc *crtc,
struct intel_encoder *encoder)

update the active DPLL for a CRTC/encoder

Parameters

struct intel_atomic_state *state

atomic state

struct intel_crtc *crtc

the CRTC for which to update the active DPLL

struct intel_encoder *encoder

encoder determining the type of port DPLL

Description

Update the active DPLL for the given **crtc/encoder** in **crtc**'s atomic state, from the port DPLLs reserved previously by [intel_reserve_shared_dpll\(\)](#). The DPLL selected will be based on the current mode of the encoder's port.

int **intel_dpll_get_freq**(struct drm_i915_private *i915, const struct [intel_shared_dpll](#) *pll,
const struct intel_dpll_hw_state *pll_state)

calculate the DPLL's output frequency

Parameters

struct drm_i915_private *i915

i915 device

const struct intel_shared_dpll *pll

DPLL for which to calculate the output frequency

const struct intel_dpll_hw_state *pll_state

DPLL state from which to calculate the output frequency

Description

Return the output frequency corresponding to **pll**'s passed in **pll_state**.

bool **intel_dpll_get_hw_state**(struct drm_i915_private *i915, struct [intel_shared_dpll](#) *pll,
struct intel_dpll_hw_state *hw_state)

readout the DPLL's hardware state

Parameters

struct drm_i915_private *i915

i915 device

struct intel_shared_dpll *pll

DPLL for which to calculate the output frequency

struct intel_dpll_hw_state *hw_state

DPLL's hardware state

Description

Read out **pll**'s hardware state into **hw_state**.

void **intel_dpll_dump_hw_state**(struct drm_i915_private *dev_priv, const struct
intel_dpll_hw_state *hw_state)

write hw_state to dmesg

Parameters

struct drm_i915_private *dev_priv

i915 drm device

const struct intel_dpll_hw_state *hw_state
hw state to be written to the log

Description

Write the relevant values in **hw_state** to dmesg using `drm_dbg_kms`.

enum **intel_dpll_id**
possible DPLL ids

Constants

DPLL_ID_PRIVATE
non-shared dpll in use

DPLL_ID_PCH_PLL_A
DPLL A in ILK, SNB and IVB

DPLL_ID_PCH_PLL_B
DPLL B in ILK, SNB and IVB

DPLL_ID_WRPLL1
HSW and BDW WRPLL1

DPLL_ID_WRPLL2
HSW and BDW WRPLL2

DPLL_ID_SPLL
HSW and BDW SPLL

DPLL_ID_LCPLL_810
HSW and BDW 0.81 GHz LCPLL

DPLL_ID_LCPLL_1350
HSW and BDW 1.35 GHz LCPLL

DPLL_ID_LCPLL_2700
HSW and BDW 2.7 GHz LCPLL

DPLL_ID_SKL_DPLL0
SKL and later DPLL0

DPLL_ID_SKL_DPLL1
SKL and later DPLL1

DPLL_ID_SKL_DPLL2
SKL and later DPLL2

DPLL_ID_SKL_DPLL3
SKL and later DPLL3

DPLL_ID_ICL_DPLL0
ICL/TGL combo PHY DPLL0

DPLL_ID_ICL_DPLL1
ICL/TGL combo PHY DPLL1

DPLL_ID_EHL_DPLL4
EHL combo PHY DPLL4

DPLL_ID_ICL_TBTPLL
ICL/TGL TBT PLL

DPLL_ID_ICL_MGPLL1

ICL MG PLL 1 port 1 (C),
TGL TC PLL 1 port 1 (TC1)

DPLL_ID_ICL_MGPLL2

ICL MG PLL 1 port 2 (D)
TGL TC PLL 1 port 2 (TC2)

DPLL_ID_ICL_MGPLL3

ICL MG PLL 1 port 3 (E)
TGL TC PLL 1 port 3 (TC3)

DPLL_ID_ICL_MGPLL4

ICL MG PLL 1 port 4 (F)
TGL TC PLL 1 port 4 (TC4)

DPLL_ID_TGL_MGPLL5

TGL TC PLL port 5 (TC5)

DPLL_ID_TGL_MGPLL6

TGL TC PLL port 6 (TC6)

DPLL_ID_DG1_DPLL0

DG1 combo PHY DPLL0

DPLL_ID_DG1_DPLL1

DG1 combo PHY DPLL1

DPLL_ID_DG1_DPLL2

DG1 combo PHY DPLL2

DPLL_ID_DG1_DPLL3

DG1 combo PHY DPLL3

Description

Enumeration of possible IDs for a DPLL. Real shared dpll ids must be ≥ 0 .

struct **intel_shared_dpll_state**

hold the DPLL atomic state

Definition:

```
struct intel_shared_dpll_state {
    u8 pipe_mask;
    struct intel_dpll_hw_state hw_state;
};
```

Members**pipe_mask**

mask of pipes using this DPLL, active or not

hw_state

hardware configuration for the DPLL stored in struct intel_dpll_hw_state.

Description

This structure holds an atomic state for the DPLL, that can represent either its current state (in struct *intel_shared_dpll*) or a desired future state which would be applied by an atomic mode set (stored in a struct *intel_atomic_state*).

See also *intel_reserve_shared_dplls()* and *intel_release_shared_dplls()*.

struct **dpll_info**

display PLL platform specific info

Definition:

```
struct dpll_info {
    const char *name;
    const struct intel_shared_dpll_funcs *funcs;
    enum intel_dpll_id id;
#define INTEL_DPLL_ALWAYS_ON    (1 << 0);
    u32 flags;
};
```

Members

name

DPLL name; used for logging

funcs

platform specific hooks

id

unique identifier for this DPLL; should match the index in the dev_priv->shared_dplls array

flags

INTEL_DPLL_ALWAYS_ON

Inform the state checker that the DPLL is kept enabled even if not in use by any CRTC.

struct **intel_shared_dpll**

display PLL with tracked state and users

Definition:

```
struct intel_shared_dpll {
    struct intel_shared_dpll_state state;
    u8 active_mask;
    bool on;
    const struct dpll_info *info;
    intel_wakeref_t wakeref;
};
```

Members

state

Store the state for the pll, including its hw state and CRTC's using it.

active_mask

mask of active pipes (i.e. DPMS on) using this DPLL

on

is the PLL actually active? Disabled during modeset

info

platform specific info

wakeref

In some platforms a device-level runtime pm reference may need to be grabbed to disable DC states while this DPLL is enabled

Display State Buffer

A DSB (Display State Buffer) is a queue of MMIO instructions in the memory which can be offloaded to DSB HW in Display Controller. DSB HW is a DMA engine that can be programmed to download the DSB from memory. It allows driver to batch submit display HW programming. This helps to reduce loading time and CPU activity, thereby making the context switch faster. DSB Support added from Gen12 Intel graphics based platform.

DSB's can access only the pipe, plane, and transcoder Data Island Packet registers.

DSB HW can support only register writes (both indexed and direct MMIO writes). There are no registers reads possible with DSB HW engine.

void **intel_dsb_reg_write**(struct intel_dsb *dsb, i915_reg_t reg, u32 val)

Emit register write to the DSB context

Parameters

struct intel_dsb *dsb

DSB context

i915_reg_t reg

register address.

u32 val

value.

Description

This function is used for writing register-value pair in command buffer of DSB.

void **intel_dsb_commit**(struct intel_dsb *dsb, bool wait_for_vblank)

Trigger workload execution of DSB.

Parameters

struct intel_dsb *dsb

DSB context

bool wait_for_vblank

wait for vblank before executing

Description

This function is used to do actual write to hardware using DSB.

struct intel_dsb ***intel_dsb_prepare**(struct intel_crtc *crtc, unsigned int max_cmds)

Allocate, pin and map the DSB command buffer.

Parameters

struct intel_crtc *crtc
the CRTC

unsigned int max_cmds
number of commands we need to fit into command buffer

Description

This function prepare the command buffer which is used to store dsb instructions with data.

Return

DSB context, NULL on failure

void **intel_dsb_cleanup**(struct intel_dsb *dsb)
To cleanup DSB context.

Parameters

struct intel_dsb *dsb
DSB context

Description

This function cleanup the DSB context by unpinning and releasing the VMA object associated with it.

10.2.3 GT Programming

Multicast/Replicated (MCR) Registers

Some GT registers are designed as “multicast” or “replicated” registers: multiple instances of the same register share a single MMIO offset. MCR registers are generally used when the hardware needs to potentially track independent values of a register per hardware unit (e.g., per-subslice, per-L3bank, etc.). The specific types of replication that exist vary per-platform.

MMIO accesses to MCR registers are controlled according to the settings programmed in the platform’s MCR_SELECTOR register(s). MMIO writes to MCR registers can be done in either a (i.e., a single write updates all instances of the register to the same value) or unicast (a write updates only one specific instance). Reads of MCR registers always operate in a unicast manner regardless of how the multicast/unicast bit is set in MCR_SELECTOR. Selection of a specific MCR instance for unicast operations is referred to as “steering.”

If MCR register operations are steered toward a hardware unit that is fused off or currently powered down due to power gating, the MMIO operation is “terminated” by the hardware. Terminated read operations will return a value of zero and terminated unicast write operations will be silently ignored.

void **intel_gt_mcr_lock**(struct intel_gt *gt, unsigned long *flags)
Acquire MCR steering lock

Parameters

struct intel_gt *gt
GT structure

unsigned long *flags
storage to save IRQ flags to

Description

Performs locking to protect the steering for the duration of an MCR operation. On MTL and beyond, a hardware lock will also be taken to serialize access not only for the driver, but also for external hardware and firmware agents.

Context

Takes `gt->mcr_lock`. `uncore->lock` should *not* be held when this function is called, although it may be acquired after this function call.

void **intel_gt_mcr_unlock**(struct intel_gt *gt, unsigned long flags)
Release MCR steering lock

Parameters

struct intel_gt *gt
GT structure

unsigned long flags
IRQ flags to restore

Description

Releases the lock acquired by [*intel_gt_mcr_lock\(\)*](#).

Context

Releases `gt->mcr_lock`

u32 **intel_gt_mcr_read**(struct intel_gt *gt, i915_mcr_reg_t reg, int group, int instance)
read a specific instance of an MCR register

Parameters

struct intel_gt *gt
GT structure

i915_mcr_reg_t reg
the MCR register to read

int group
the MCR group

int instance
the MCR instance

Context

Takes and releases `gt->mcr_lock`

Description

Returns the value read from an MCR register after steering toward a specific group/instance.

void **intel_gt_mcr_unicast_write**(struct intel_gt *gt, i915_mcr_reg_t reg, u32 value, int group, int instance)
write a specific instance of an MCR register

Parameters

struct intel_gt *gt
GT structure

i915_mcr_reg_t reg

the MCR register to write

u32 value

value to write

int group

the MCR group

int instance

the MCR instance

Description

Write an MCR register in unicast mode after steering toward a specific group/instance.

Context

Calls a function that takes and releases `gt->mcr_lock`

void **intel_gt_mcr_multicast_write**(struct intel_gt *gt, i915_mcr_reg_t reg, u32 value)

write a value to all instances of an MCR register

Parameters

struct intel_gt *gt

GT structure

i915_mcr_reg_t reg

the MCR register to write

u32 value

value to write

Description

Write an MCR register in multicast mode to update all instances.

Context

Takes and releases `gt->mcr_lock`

void **intel_gt_mcr_multicast_write_fw**(struct intel_gt *gt, i915_mcr_reg_t reg, u32 value)

write a value to all instances of an MCR register

Parameters

struct intel_gt *gt

GT structure

i915_mcr_reg_t reg

the MCR register to write

u32 value

value to write

Description

Write an MCR register in multicast mode to update all instances. This function assumes the caller is already holding any necessary forcewake domains; use [`intel_gt_mcr_multicast_write\(\)`](#) in cases where forcewake should be obtained automatically.

Context

The caller must hold `gt->mcr_lock`.

u32 **intel_gt_mcr_multicast_rmw**(struct intel_gt *gt, i915_mcr_reg_t reg, u32 clear, u32 set)

Performs a multicast RMW operations

Parameters

struct intel_gt *gt

GT structure

i915_mcr_reg_t reg

the MCR register to read and write

u32 clear

bits to clear during RMW

u32 set

bits to set during RMW

Description

Performs a read-modify-write on an MCR register in a multicast manner. This operation only makes sense on MCR registers where all instances are expected to have the same value. The read will target any non-terminated instance and the write will be applied to all instances.

This function assumes the caller is already holding any necessary forcewake domains; use [intel_gt_mcr_multicast_rmw\(\)](#) in cases where forcewake should be obtained automatically.

Returns the old (unmodified) value read.

Context

Calls functions that take and release `gt->mcr_lock`

void **intel_gt_mcr_get_nonterminated_steering**(struct intel_gt *gt, i915_mcr_reg_t reg, u8 *group, u8 *instance)

find group/instance values that will steer a register to a non-terminated instance

Parameters

struct intel_gt *gt

GT structure

i915_mcr_reg_t reg

register for which the steering is required

u8 *group

return variable for group steering

u8 *instance

return variable for instance steering

Description

This function returns a group/instance pair that is guaranteed to work for read steering of the given register. Note that a value will be returned even if the register is not replicated and therefore does not actually require steering.

u32 **intel_gt_mcr_read_any_fw**(struct intel_gt *gt, i915_mcr_reg_t reg)

reads one instance of an MCR register

Parameters

struct intel_gt *gt

GT structure

i915_mcr_reg_t reg

register to read

Description

Reads a GT MCR register. The read will be steered to a non-terminated instance (i.e., one that isn't fused off or powered down by power gating). This function assumes the caller is already holding any necessary forcewake domains; use [intel_gt_mcr_read_any\(\)](#) in cases where forcewake should be obtained automatically.

Returns the value from a non-terminated instance of **reg**.

Context

The caller must hold gt->mcr_lock.

u32 **intel_gt_mcr_read_any**(struct intel_gt *gt, i915_mcr_reg_t reg)

reads one instance of an MCR register

Parameters

struct intel_gt *gt

GT structure

i915_mcr_reg_t reg

register to read

Description

Reads a GT MCR register. The read will be steered to a non-terminated instance (i.e., one that isn't fused off or powered down by power gating).

Returns the value from a non-terminated instance of **reg**.

Context

Calls a function that takes and releases gt->mcr_lock.

void **intel_gt_mcr_get_ss_steering**(struct intel_gt *gt, unsigned int dss, unsigned int *group, unsigned int *instance)

returns the group/instance steering for a SS

Parameters

struct intel_gt *gt

GT structure

unsigned int dss

DSS ID to obtain steering for

unsigned int *group

pointer to storage for steering group ID

unsigned int *instance

pointer to storage for steering instance ID

Description

Returns the steering IDs (via the **group** and **instance** parameters) that correspond to a specific subslice/DSS ID.

int **intel_gt_mcr_wait_for_reg**(struct intel_gt *gt, i915_mcr_reg_t reg, u32 mask, u32 value, unsigned int fast_timeout_us, unsigned int slow_timeout_ms)

wait until MCR register matches expected state

Parameters

struct intel_gt *gt

GT structure

i915_mcr_reg_t reg

the register to read

u32 mask

mask to apply to register value

u32 value

value to wait for

unsigned int fast_timeout_us

fast timeout in microsecond for atomic/tight wait

unsigned int slow_timeout_ms

slow timeout in millisecond

Description

This routine waits until the target register **reg** contains the expected **value** after applying the **mask**, i.e. it waits until

```
(intel_gt_mcr_read_any_fw(gt, reg) & mask) == value
```

Otherwise, the wait will timeout after **slow_timeout_ms** milliseconds. For atomic context **slow_timeout_ms** must be zero and **fast_timeout_us** must be not larger than 20,000 microseconds.

This function is basically an MCR-friendly version of `__intel_wait_for_register_fw()`. Generally this function will only be used on GAM registers which are a bit special --- although they're MCR registers, reads (e.g., waiting for status updates) are always directed to the primary instance.

Note that this routine assumes the caller holds forcewake asserted, it is not suitable for very long waits.

Context

Calls a function that takes and releases gt->mcr_lock

Return

0 if the register matches the desired condition, or -ETIMEDOUT.

10.2.4 Memory Management and Command Submission

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

Intel GPU Basics

An Intel GPU has multiple engines. There are several engine types.

- RCS engine is for rendering 3D and performing compute, this is named *I915_EXEC_RENDER* in user space.
- BCS is a blitting (copy) engine, this is named *I915_EXEC_BLT* in user space.
- VCS is a video encode and decode engine, this is named *I915_EXEC_BSD* in user space
- VECS is video enhancement engine, this is named *I915_EXEC_VEBOX* in user space.
- The enumeration *I915_EXEC_DEFAULT* does not refer to specific engine; instead it is to be used by user space to specify a default rendering engine (for 3D) that may or may not be the same as RCS.

The Intel GPU family is a family of integrated GPU's using Unified Memory Access. For having the GPU "do work", user space will feed the GPU batch buffers via one of the ioctls *DRM_IOCTL_I915_GEM_EXECBUFFER2* or *DRM_IOCTL_I915_GEM_EXECBUFFER2_WR*. Most such batchbuffers will instruct the GPU to perform work (for example rendering) and that work needs memory from which to read and memory to which to write. All memory is encapsulated within GEM buffer objects (usually created with the ioctl *DRM_IOCTL_I915_GEM_CREATE*). An ioctl providing a batchbuffer for the GPU to create will also list all GEM buffer objects that the batchbuffer reads and/or writes. For implementation details of memory management see [GEM BO Management Implementation Details](#).

The i915 driver allows user space to create a context via the ioctl *DRM_IOCTL_I915_GEM_CONTEXT_CREATE* which is identified by a 32-bit integer. Such a context should be viewed by user-space as -loosely- analogous to the idea of a CPU process of an operating system. The i915 driver guarantees that commands issued to a fixed context are to be executed so that writes of a previously issued command are seen by reads of following commands. Actions issued between different contexts (even if from the same file descriptor) are NOT given that guarantee and the only way to synchronize across contexts (even from the same file descriptor) is through the use of fences. At least as far back as Gen4, also have that a context carries with it a GPU HW context; the HW context is essentially (most of at least) the state of a GPU. In addition to the ordering guarantees, the kernel will restore GPU state via HW context when commands are issued to a context, this saves user space the need to restore (most of at least) the GPU state at the start of each batchbuffer. The non-deprecated ioctls to submit batchbuffer work can pass that ID (in the lower bits of *drm_i915_gem_execbuffer2::rsvd1*) to identify what context to use with the command.

The GPU has its own memory management and address space. The kernel driver maintains the memory translation table for the GPU. For older GPUs (i.e. those before Gen8), there is a single global such translation table, a global Graphics Translation Table (GTT). For newer generation GPUs each context has its own translation table, called Per-Process Graphics Translation Table (PPGTT). Of important note, is that although PPGTT is named per-process it is actually per context. When user space submits a batchbuffer, the kernel walks the list of GEM buffer objects used by the batchbuffer and guarantees that not only is the memory of each such GEM buffer object resident but it is also present in the (PP)GTT. If the GEM buffer object is not yet placed in the (PP)GTT, then it is given an address. Two consequences of this are: the kernel needs to

edit the batchbuffer submitted to write the correct value of the GPU address when a GEM BO is assigned a GPU address and the kernel might evict a different GEM BO from the (PP)GTT to make address room for another GEM BO. Consequently, the ioctls submitting a batchbuffer for execution also include a list of all locations within buffers that refer to GPU-addresses so that the kernel can edit the buffer correctly. This process is dubbed relocation.

Locking Guidelines

Note: This is a description of how the locking should be after refactoring is done. Does not necessarily reflect what the locking looks like while WIP.

1. All locking rules and interface contracts with cross-driver interfaces (dma-buf, dma_fence) need to be followed.
2. No struct_mutex anywhere in the code
3. dma_resv will be the outermost lock (when needed) and ww_acquire_ctx is to be hoisted at highest level and passed down within i915_gem_ctx in the call chain
4. While holding lru/memory manager (buddy, drm_mm, whatever) locks system memory allocations are not allowed
 - Enforce this by priming lockdep (with fs_reclaim). If we allocate memory while holding these locks we get a rehash of the shrinker vs. struct_mutex saga, and that would be real bad.
5. Do not nest different lru/memory manager locks within each other. Take them in turn to update memory allocations, relying on the object's dma_resv ww_mutex to serialize against other operations.
6. The suggestion for lru/memory managers locks is that they are small enough to be spinlocks.
7. All features need to come with exhaustive kernel selftests and/or IGT tests when appropriate
8. All LMEM uAPI paths need to be fully restartable (_interruptible()) for all locks/waits/sleeps)
 - Error handling validation through signal injection. Still the best strategy we have for validating GEM uAPI corner cases. Must be excessively used in the IGT, and we need to check that we really have full path coverage of all error cases.
 - -EDEADLK handling with ww_mutex

GEM BO Management Implementation Details

A VMA represents a GEM BO that is bound into an address space. Therefore, a VMA's presence cannot be guaranteed before binding, or after unbinding the object into/from the address space.

To make things as simple as possible (ie. no refcounting), a VMA's lifetime will always be \leq an object's lifetime. So object refcounting should cover us.

Buffer Object Eviction

This section documents the interface functions for evicting buffer objects to make space available in the virtual gpu address spaces. Note that this is mostly orthogonal to shrinking buffer objects caches, which has the goal to make main memory (shared with the gpu through the unified memory architecture) available.

```
int i915_gem_evict_something(struct i915_address_space *vm, struct i915_gem_ww_ctx
                             *ww, u64 min_size, u64 alignment, unsigned long color, u64
                             start, u64 end, unsigned flags)
```

Evict vmas to make room for binding a new one

Parameters

struct i915_address_space *vm
address space to evict from

struct i915_gem_ww_ctx *ww
An optional struct i915_gem_ww_ctx.

u64 min_size
size of the desired free space

u64 alignment
alignment constraint of the desired free space

unsigned long color
color for the desired space

u64 start
start (inclusive) of the range from which to evict objects

u64 end
end (exclusive) of the range from which to evict objects

unsigned flags
additional flags to control the eviction algorithm

Description

This function will try to evict vmas until a free space satisfying the requirements is found. Callers must check first whether any such hole exists already before calling this function.

This function is used by the object/vma binding code.

Since this function is only used to free up virtual address space it only ignores pinned vmas, and not object where the backing storage itself is pinned. Hence `obj->pages_pin_count` does not protect against eviction.

To clarify: This is for freeing up virtual address space, not for freeing memory in e.g. the shrinker.

```
int i915_gem_evict_for_node(struct i915_address_space *vm, struct i915_gem_ww_ctx *ww,
                             struct drm_mm_node *target, unsigned int flags)
```

Evict vmas to make room for binding a new one

Parameters

struct i915_address_space *vm
address space to evict from

struct i915_gem_ww_ctx *ww

An optional struct i915_gem_ww_ctx.

struct drm_mm_node *target

range (and color) to evict for

unsigned int flags

additional flags to control the eviction algorithm

Description

This function will try to evict vmas that overlap the target node.

To clarify: This is for freeing up virtual address space, not for freeing memory in e.g. the shrinker.

int **i915_gem_evict_vm**(struct i915_address_space *vm, struct i915_gem_ww_ctx *ww, struct drm_i915_gem_object **busy_bo)

Evict all idle vmas from a vm

Parameters

struct i915_address_space *vm

Address space to cleanse

struct i915_gem_ww_ctx *ww

An optional struct i915_gem_ww_ctx. If not NULL, i915_gem_evict_vm will be able to evict vma's locked by the ww as well.

struct drm_i915_gem_object **busy_bo

Optional pointer to struct drm_i915_gem_object. If not NULL, then in the event [i915_gem_evict_vm\(\)](#) is unable to trylock an object for eviction, then **busy_bo** will point to it. -EBUSY is also returned. The caller must drop the vm->mutex, before trying again to acquire the contended lock. The caller also owns a reference to the object.

Description

This function evicts all vmas from a vm.

This is used by the execbuf code as a last-ditch effort to defragment the address space.

To clarify: This is for freeing up virtual address space, not for freeing memory in e.g. the shrinker.

Buffer Object Memory Shrinking

This section documents the interface function for shrinking memory usage of buffer object caches. Shrinking is used to make main memory available. Note that this is mostly orthogonal to evicting buffer objects, which has the goal to make space in gpu virtual address spaces.

unsigned long **i915_gem_shrink**(struct i915_gem_ww_ctx *ww, struct drm_i915_private *i915, unsigned long target, unsigned long *nr_scanned, unsigned int shrink)

Shrink buffer object caches

Parameters

struct i915_gem_ww_ctx *ww
i915 gem ww acquire ctx, or NULL

struct drm_i915_private *i915
i915 device

unsigned long target
amount of memory to make available, in pages

unsigned long *nr_scanned
optional output for number of pages scanned (incremental)

unsigned int shrink
control flags for selecting cache types

Description

This function is the main interface to the shrinker. It will try to release up to **target** pages of main memory backing storage from buffer objects. Selection of the specific caches can be done with **flags**. This is e.g. useful when purgeable objects should be removed from caches preferentially.

Note that it's not guaranteed that released amount is actually available as free system memory - the pages might still be in-used due to other reasons (like cpu mmaps) or the mm core has reused them before we could grab them. Therefore code that needs to explicitly shrink buffer objects caches (e.g. to avoid deadlocks in memory reclaim) must fall back to [i915_gem_shrink_all\(\)](#).

Also note that any kind of pinning (both per-vma address space pins and backing storage pins at the buffer object level) result in the shrinker code having to skip the object.

Return

The number of pages of backing storage actually released.

unsigned long **i915_gem_shrink_all**(struct drm_i915_private *i915)
Shrink buffer object caches completely

Parameters

struct drm_i915_private *i915
i915 device

Description

This is a simple wrapper around [i915_gem_shrink\(\)](#) to aggressively shrink all caches completely. It also first waits for and retires all outstanding requests to also be able to release backing storage for active objects.

This should only be used in code to intentionally quiescent the gpu or as a last-ditch effort when memory seems to have run out.

Return

The number of pages of backing storage actually released.

void **i915_gem_object_make_unshrinkable**(struct drm_i915_gem_object *obj)
Hide the object from the shrinker. By default all object types that support shrinking(see IS_SHRINKABLE), will also make the object visible to the shrinker after allocating the system memory pages.

Parameters

struct drm_i915_gem_object *obj
The GEM object.

Description

This is typically used for special kernel internal objects that can't be easily processed by the shrinker, like if they are perma-pinned.

void __i915_gem_object_make_shrinkable(struct drm_i915_gem_object *obj)
Move the object to the tail of the shrinkable list. Objects on this list might be swapped out. Used with WILLNEED objects.

Parameters

struct drm_i915_gem_object *obj
The GEM object.

Description

DO NOT USE. This is intended to be called on very special objects that don't yet have mm.pages, but are guaranteed to have potentially reclaimable pages underneath.

void __i915_gem_object_make_purgeable(struct drm_i915_gem_object *obj)
Move the object to the tail of the purgeable list. Objects on this list might be swapped out. Used with DONTNEED objects.

Parameters

struct drm_i915_gem_object *obj
The GEM object.

Description

DO NOT USE. This is intended to be called on very special objects that don't yet have mm.pages, but are guaranteed to have potentially reclaimable pages underneath.

void i915_gem_object_make_shrinkable(struct drm_i915_gem_object *obj)
Move the object to the tail of the shrinkable list. Objects on this list might be swapped out. Used with WILLNEED objects.

Parameters

struct drm_i915_gem_object *obj
The GEM object.

Description

MUST only be called on objects which have backing pages.

MUST be balanced with previous call to [*i915_gem_object_make_unshrinkable\(\)*](#).

void i915_gem_object_make_purgeable(struct drm_i915_gem_object *obj)
Move the object to the tail of the purgeable list. Used with DONTNEED objects. Unlike with shrinkable objects, the shrinker will attempt to discard the backing pages, instead of trying to swap them out.

Parameters

struct drm_i915_gem_object *obj
The GEM object.

Description

MUST only be called on objects which have backing pages.

MUST be balanced with previous call to `i915_gem_object_make_unshrinkable()`.

Batchbuffer Parsing

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

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

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

First, commands which are explicitly defined as privileged or which should only be used by the kernel driver. The parser rejects such commands

Second, commands which access registers. To support correct/enhanced userspace functionality, particularly certain OpenGL extensions, the parser provides a whitelist of registers which userspace may safely access

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

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

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

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

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

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

```
int intel_engine_init_cmd_parser(struct intel_engine_cs *engine)
    set cmd parser related fields for an engine
```

Parameters

struct intel_engine_cs *engine

the engine to initialize

Description

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

void intel_engine_cleanup_cmd_parser(struct intel_engine_cs *engine)

clean up cmd parser related fields

Parameters

struct intel_engine_cs *engine

the engine to clean up

Description

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

int intel_engine_cmd_parser(struct intel_engine_cs *engine, struct i915_vma *batch, unsigned long batch_offset, unsigned long batch_length, struct i915_vma *shadow, bool trampoline)

parse a batch buffer for privilege violations

Parameters

struct intel_engine_cs *engine

the engine on which the batch is to execute

struct i915_vma *batch

the batch buffer in question

unsigned long batch_offset

byte offset in the batch at which execution starts

unsigned long batch_length

length of the commands in batch_obj

struct i915_vma *shadow

validated copy of the batch buffer in question

bool trampoline

true if we need to trampoline into privileged execution

Description

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

Return

non-zero if the parser finds violations or otherwise fails; -EACCES if the batch appears legal but should use hardware parsing

int i915_cmd_parser_get_version(struct drm_i915_private *dev_priv)

get the cmd parser version number

Parameters

struct drm_i915_private *dev_priv

i915 device private

Description

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

Return

the current version number of the cmd parser

User Batchbuffer Execution

struct **i915_gem_engines**

A set of engines

Definition:

```
struct i915_gem_engines {
    union {
        struct list_head link;
        struct rcu_head rcu;
    };
    struct i915_sw_fence fence;
    struct i915_gem_context *ctx;
    unsigned int num_engines;
    struct intel_context *engines[];
};
```

Members

{unnamed_union}

anonymous

link

Link in i915_gem_context::stale::engines

rcu

RCU to use when freeing

fence

Fence used for delayed destruction of engines

ctx

i915_gem_context backpointer

num_engines

Number of engines in this set

engines

Array of engines

struct **i915_gem_engines_iter**

Iterator for an i915_gem_engines set

Definition:

```
struct i915_gem_engines_iter {
    unsigned int idx;
```



```
const struct i915_gem_engines *engines;
};
```

Members

idx

Index into i915_gem_engines::engines

engines

Engine set being iterated

enum i915_gem_engine_type

Describes the type of an i915_gem_proto_engine

Constants

I915_GEM_ENGINE_TYPE_INVALID

An invalid engine

I915_GEM_ENGINE_TYPE_PHYSICAL

A single physical engine

I915_GEM_ENGINE_TYPE_BALANCED

A load-balanced engine set

I915_GEM_ENGINE_TYPE_PARALLEL

A parallel engine set

struct i915_gem_proto_engine

prototype engine

Definition:

```
struct i915_gem_proto_engine {
    enum i915_gem_engine_type type;
    struct intel_engine_cs *engine;
    unsigned int num_siblings;
    unsigned int width;
    struct intel_engine_cs **siblings;
    struct intel_sseu sseu;
};
```

Members

type

Type of this engine

engine

Engine, for physical

num_siblings

Number of balanced or parallel siblings

width

Width of each sibling

siblings

Balanced siblings or num_siblings * width for parallel

sseu

Client-set SSEU parameters

Description

This struct describes an engine that a context may contain. Engines have four types:

- `I915_GEM_ENGINE_TYPE_INVALID`: Invalid engines can be created but they show up as a NULL in `i915_gem_engines::engines[i]` and any attempt to use them by the user results in `-EINVAL`. They are also useful during proto-context construction because the client may create invalid engines and then set them up later as virtual engines.
- `I915_GEM_ENGINE_TYPE_PHYSICAL`: A single physical engine, described by `i915_gem_proto_engine::engine`.
- `I915_GEM_ENGINE_TYPE_BALANCED`: A load-balanced engine set, described by `i915_gem_proto_engine::num_siblings` and `i915_gem_proto_engine::siblings`.
- `I915_GEM_ENGINE_TYPE_PARALLEL`: A parallel submission engine set, described by `i915_gem_proto_engine::width`, `i915_gem_proto_engine::num_siblings`, and `i915_gem_proto_engine::siblings`.

struct **i915_gem_proto_context**
 prototype context

Definition:

```
struct i915_gem_proto_context {
    struct i915_address_space *vm;
    unsigned long user_flags;
    struct i915_sched_attr sched;
    int num_user_engines;
    struct i915_gem_proto_engine *user_engines;
    struct intel_sseu legacy_rcs_sseu;
    bool single_timeline;
    bool uses_protected_content;
    intel_wakeref_t pxp_wakeref;
};
```

Members

vm

See [*i915_gem_context.vm*](#)

user_flags

See [*i915_gem_context.user_flags*](#)

sched

See [*i915_gem_context.sched*](#)

num_user_engines

Number of user-specified engines or -1

user_engines

User-specified engines

legacy_rcs_sseu

Client-set SSEU parameters for the legacy RCS

single_timeline

See *See [i915_gem_context.syncobj](#)*

uses_protected_content

See *See [i915_gem_context.uses_protected_content](#)*

pxp_wakeref

See *See [i915_gem_context.pxp_wakeref](#)*

Description

The *struct i915_gem_proto_context* represents the creation parameters for a *struct i915_gem_context*. This is used to gather parameters provided either through creation flags or via SET_CONTEXT_PARAM so that, when we create the final i915_gem_context, those parameters can be immutable.

The context uAPI allows for two methods of setting context parameters: SET_CONTEXT_PARAM and CONTEXT_CREATE_EXT_SETPARAM. The former is allowed to be called at any time while the later happens as part of GEM_CONTEXT_CREATE. When these were initially added, Currently, everything settable via one is settable via the other. While some params are fairly simple and setting them on a live context is harmless such the context priority, others are far trickier such as the VM or the set of engines. To avoid some truly nasty race conditions, we don't allow setting the VM or the set of engines on live contexts.

The way we dealt with this without breaking older userspace that sets the VM or engine set via SET_CONTEXT_PARAM is to delay the creation of the actual context until after the client is done configuring it with SET_CONTEXT_PARAM. From the perspective of the client, it has the same u32 context ID the whole time. From the perspective of i915, however, it's an i915_gem_proto_context right up until the point where we attempt to do something which the proto-context can't handle at which point the real context gets created.

This is accomplished via a little xarray dance. When GEM_CONTEXT_CREATE is called, we create a proto-context, reserve a slot in context_xa but leave it NULL, the proto-context in the corresponding slot in proto_context_xa. Then, whenever we go to look up a context, we first check context_xa. If it's there, we return the i915_gem_context and we're done. If it's not, we look in proto_context_xa and, if we find it there, we create the actual context and kill the proto-context.

At the time we made this change (April, 2021), we did a fairly complete audit of existing userspace to ensure this wouldn't break anything:

- Mesa/i965 didn't use the engines or VM APIs at all
- Mesa/ANV used the engines API but via CONTEXT_CREATE_EXT_SETPARAM and didn't use the VM API.
- Mesa/iris didn't use the engines or VM APIs at all
- The open-source compute-runtime didn't yet use the engines API but did use the VM API via SET_CONTEXT_PARAM. However, CONTEXT_SETPARAM was always the second ioctl on that context, immediately following GEM_CONTEXT_CREATE.
- The media driver sets engines and bonding/balancing via SET_CONTEXT_PARAM. However, CONTEXT_SETPARAM to set the VM was always the second ioctl on that context, immediately following GEM_CONTEXT_CREATE and setting engines immediately followed that.

In order for this dance to work properly, any modification to an `i915_gem_proto_context` that is exposed to the client via `drm_i915_file_private::proto_context_xa` must be guarded by `drm_i915_file_private::proto_context_lock`. The exception is when a proto-context has not yet been exposed such as when handling `CONTEXT_CREATE_SET_PARAM` during `GEM_CONTEXT_CREATE`.

struct **i915_gem_context**

client state

Definition:

```
struct i915_gem_context {
    struct drm_i915_private *i915;
    struct drm_i915_file_private *file_priv;
    struct i915_gem_engines __rcu *engines;
    struct mutex engines_mutex;
    struct drm_syncobj *syncobj;
    struct i915_address_space *vm;
    struct pid *pid;
    struct list_head link;
    struct i915_drm_client *client;
    struct list_head client_link;
    struct kref ref;
    struct work_struct release_work;
    struct rcu_head rcu;
    unsigned long user_flags;
#define UCONTEXT_NO_ERROR_CAPTURE        1;
#define UCONTEXT_BANNABLE                2;
#define UCONTEXT_RECOVERABLE            3;
#define UCONTEXT_PERSISTENCE            4;
    unsigned long flags;
#define CONTEXT_CLOSED                    0;
#define CONTEXT_USER_ENGINES            1;
    bool uses_protected_content;
    intel_wakeref_t pxp_wakeref;
    struct mutex mutex;
    struct i915_sched_attr sched;
    atomic_t guilty_count;
    atomic_t active_count;
    unsigned long hang_timestamp[2];
#define CONTEXT_FAST_HANG_JIFFIES (120 * HZ) ;
    u8 remap_slice;
    struct radix_tree_root handles_vma;
    struct mutex lut_mutex;
    char name[TASK_COMM_LEN + 8];
    struct {
        spinlock_t lock;
        struct list_head engines;
    } stale;
};
```

Members

i915

i915 device backpointer

file_priv

owning file descriptor

engines

list of stale engines

engines_mutex

guards writes to engines

syncobj

Shared timeline syncobj

When the SHARED_TIMELINE flag is set on context creation, we emulate a single timeline across all engines using this syncobj. For every execbuffer2 call, this syncobj is used as both an in- and out-fence. Unlike the real intel_timeline, this doesn't provide perfect atomic in-order guarantees if the client races with itself by calling execbuffer2 twice concurrently. However, if userspace races with itself, that's not likely to yield well-defined results anyway so we choose to not care.

vm

unique address space (GTT)

In full-ppgtt mode, each context has its own address space ensuring complete separation of one client from all others.

In other modes, this is a NULL pointer with the expectation that the caller uses the shared global GTT.

pid

process id of creator

Note that who created the context may not be the principle user, as the context may be shared across a local socket. However, that should only affect the default context, all contexts created explicitly by the client are expected to be isolated.

link

place with `drm_i915_private.context_list`

client

struct `i915_drm_client`

client_link

for linking onto `i915_drm_client.ctx_list`

ref

reference count

A reference to a context is held by both the client who created it and on each request submitted to the hardware using the request (to ensure the hardware has access to the state until it has finished all pending writes). See `i915_gem_context_get()` and `i915_gem_context_put()` for access.

release_work

Work item for deferred cleanup, since `i915_gem_context_put()` tends to be called from hardirq context.

FIXME: The only real reason for this is *i915_gem_engines.fence*, all other callers are from process context and need at most some mild shuffling to pull the `i915_gem_context_put()` call out of a spinlock.

rcu

`rcu_head` for deferred freeing.

user_flags

small set of booleans controlled by the user

flags

small set of booleans

uses_protected_content

context uses PXP-encrypted objects.

This flag can only be set at ctx creation time and it's immutable for the lifetime of the context. See `I915_CONTEXT_PARAM_PROTECTED_CONTENT` in `uapi/drm/i915_drm.h` for more info on setting restrictions and expected behaviour of marked contexts.

pxp_wakeref

wakeref to keep the device awake when PXP is in use

PXP sessions are invalidated when the device is suspended, which in turns invalidates all contexts and objects using it. To keep the flow simple, we keep the device awake when contexts using PXP objects are in use. It is expected that the userspace application only uses PXP when the display is on, so taking a wakeref here shouldn't worsen our power metrics.

mutex

guards everything that isn't engines or `handles_vma`

sched

scheduler parameters

guilty_count

How many times this context has caused a GPU hang.

active_count

How many times this context was active during a GPU hang, but did not cause it.

hang_timestamp

The last time(s) this context caused a GPU hang

remap_slice

Bitmask of cache lines that need remapping

handles_vma

rbtree to look up our context specific obj/vma for the user handle. (user handles are per fd, but the binding is per vm, which may be one per context or shared with the global GTT)

lut_mutex

Locks `handles_vma`

name

arbitrary name, used for user debug

A name is constructed for the context from the creator's process name, pid and user handle in order to uniquely identify the context in messages.

stale

tracks stale engines to be destroyed

Description

The `struct i915_gem_context` represents the combined view of the driver and logical hardware state for a particular client.

Userspace submits commands to be executed on the GPU as an instruction stream within a GEM object we call a batchbuffer. These instructions may refer to other GEM objects containing auxiliary state such as kernels, samplers, render targets and even secondary batchbuffers. Userspace does not know where in the GPU memory these objects reside and so before the batchbuffer is passed to the GPU for execution, those addresses in the batchbuffer and auxiliary objects are updated. This is known as relocation, or patching. To try and avoid having to relocate each object on the next execution, userspace is told the location of those objects in this pass, but this remains just a hint as the kernel may choose a new location for any object in the future.

At the level of talking to the hardware, submitting a batchbuffer for the GPU to execute is to add content to a buffer from which the HW command streamer is reading.

1. Add a command to load the HW context. For Logical Ring Contexts, i.e. Execlists, this command is not placed on the same buffer as the remaining items.
2. Add a command to invalidate caches to the buffer.
3. Add a batchbuffer start command to the buffer; the start command is essentially a token together with the GPU address of the batchbuffer to be executed.
4. Add a pipeline flush to the buffer.
5. Add a memory write command to the buffer to record when the GPU is done executing the batchbuffer. The memory write writes the global sequence number of the request, `i915_request::global_seqno`; the i915 driver uses the current value in the register to determine if the GPU has completed the batchbuffer.
6. Add a user interrupt command to the buffer. This command instructs the GPU to issue an interrupt when the command, pipeline flush and memory write are completed.
7. Inform the hardware of the additional commands added to the buffer (by updating the tail pointer).

Processing an `execbuf` ioctl is conceptually split up into a few phases.

1. Validation - Ensure all the pointers, handles and flags are valid.
2. Reservation - Assign GPU address space for every object
3. Relocation - Update any addresses to point to the final locations
4. Serialisation - Order the request with respect to its dependencies
5. Construction - Construct a request to execute the batchbuffer
6. Submission (at some point in the future execution)

Reserving resources for the `execbuf` is the most complicated phase. We neither want to have to migrate the object in the address space, nor do we want to have to update any relocations pointing to this object. Ideally, we want to leave the object where it is and for all the existing relocations to match. If the object is given a new address, or if userspace thinks the object is elsewhere, we have to parse all the relocation entries and update the addresses. Userspace can

set the `I915_EXEC_NORELOC` flag to hint that all the target addresses in all of its objects match the value in the relocation entries and that they all match the presumed offsets given by the list of `execbuffer` objects. Using this knowledge, we know that if we haven't moved any buffers, all the relocation entries are valid and we can skip the update. (If userspace is wrong, the likely outcome is an impromptu GPU hang.) The requirement for using `I915_EXEC_NO_RELOC` are:

The addresses written in the objects must match the corresponding `reloc.presumed_offset` which in turn must match the corresponding `execobject.offset`.

Any render targets written to in the batch must be flagged with `EXEC_OBJECT_WRITE`.

To avoid stalling, `execobject.offset` should match the current address of that object within the active context.

The reservation is done in multiple phases. First we try and keep any object already bound in its current location - so as long as it meets the constraints imposed by the new `execbuffer`. Any object left unbound after the first pass is then fitted into any available idle space. If an object does not fit, all objects are removed from the reservation and the process rerun after sorting the objects into a priority order (more difficult to fit objects are tried first). Failing that, the entire VM is cleared and we try to fit the `execbuf` once last time before concluding that it simply will not fit.

A small complication to all of this is that we allow userspace not only to specify an alignment and a size for the object in the address space, but we also allow userspace to specify the exact offset. This object is simpler to place (the location is known a priori) all we have to do is make sure the space is available.

Once all the objects are in place, patching up the buried pointers to point to the final locations is a fairly simple job of walking over the relocation entry arrays, looking up the right address and rewriting the value into the object. Simple! ... The relocation entries are stored in user memory and so to access them we have to copy them into a local buffer. That copy has to avoid taking any pagefaults as they may lead back to a GEM object requiring the `struct_mutex` (i.e. recursive deadlock). So once again we split the relocation into multiple passes. First we try to do everything within an atomic context (avoid the pagefaults) which requires that we never wait. If we detect that we may wait, or if we need to fault, then we have to fallback to a slower path. The slowpath has to drop the mutex. (Can you hear alarm bells yet?) Dropping the mutex means that we lose all the state we have built up so far for the `execbuf` and we must reset any global data. However, we do leave the objects pinned in their final locations - which is a potential issue for concurrent `execbufs`. Once we have left the mutex, we can allocate and copy all the relocation entries into a large array at our leisure, reacquire the mutex, reclaim all the objects and other state and then proceed to update any incorrect addresses with the objects.

As we process the relocation entries, we maintain a record of whether the object is being written to. Using `NORELOC`, we expect userspace to provide this information instead. We also check whether we can skip the relocation by comparing the expected value inside the relocation entry with the target's final address. If they differ, we have to map the current object and rewrite the 4 or 8 byte pointer within.

Serialising an `execbuf` is quite simple according to the rules of the GEM ABI. Execution within each context is ordered by the order of submission. Writes to any GEM object are in order of submission and are exclusive. Reads from a GEM object are unordered with respect to other reads, but ordered by writes. A write submitted after a read cannot occur before the read, and similarly any read submitted after a write cannot occur before the write. Writes are ordered between engines such that only one write occurs at any time (completing any reads beforehand)

- using semaphores where available and CPU serialisation otherwise. Other GEM access obey the same rules, any write (either via mmmaps using set-domain, or via pwrite) must flush all GPU reads before starting, and any read (either using set-domain or pread) must flush all GPU writes before starting. (Note we only employ a barrier before, we currently rely on userspace not concurrently starting a new execution whilst reading or writing to an object. This may be an advantage or not depending on how much you trust userspace not to shoot themselves in the foot.) Serialisation may just result in the request being inserted into a DAG awaiting its turn, but most simple is to wait on the CPU until all dependencies are resolved.

After all of that, is just a matter of closing the request and handing it to the hardware (well, leaving it in a queue to be executed). However, we also offer the ability for batchbuffers to be run with elevated privileges so that they access otherwise hidden registers. (Used to adjust L3 cache etc.) Before any batch is given extra privileges we first must check that it contains no nefarious instructions, we check that each instruction is from our whitelist and all registers are also from an allowed list. We first copy the user's batchbuffer to a shadow (so that the user doesn't have access to it, either by the CPU or GPU as we scan it) and then parse each instruction. If everything is ok, we set a flag telling the hardware to run the batchbuffer in trusted mode, otherwise the ioctl is rejected.

Scheduling

struct **i915_sched_engine**
scheduler engine

Definition:

```
struct i915_sched_engine {
    struct kref ref;
    spinlock_t lock;
    struct list_head requests;
    struct list_head hold;
    struct tasklet_struct tasklet;
    struct i915_priolist default_priolist;
    int queue_priority_hint;
    struct rb_root_cached queue;
    bool no_priolist;
    void *private_data;
    void (*destroy)(struct kref *kref);
    bool (*disabled)(struct i915_sched_engine *sched_engine);
    void (*kick_backend)(const struct i915_request *rq, int prio);
    void (*bump_inflight_request_prio)(struct i915_request *rq, int prio);
    void (*retire_inflight_request_prio)(struct i915_request *rq);
    void (*schedule)(struct i915_request *request, const struct i915_sched_
    attr *attr);
};
```

Members

ref
reference count of schedule engine object

lock
protects requests in priority lists, requests, hold and tasklet while running

requests

list of requests inflight on this schedule engine

hold

list of ready requests, but on hold

tasklet

softirq tasklet for submission

default_priolist

priority list for I915_PRIORITY_NORMAL

queue_priority_hint

Highest pending priority.

When we add requests into the queue, or adjust the priority of executing requests, we compute the maximum priority of those pending requests. We can then use this value to determine if we need to preempt the executing requests to service the queue. However, since we may have recorded the priority of an inflight request we wanted to preempt but since completed, at the time of dequeuing the priority hint may no longer match the highest available request priority.

queue

queue of requests, in priority lists

no_priolist

priority lists disabled

private_data

private data of the submission backend

destroy

destroy schedule engine / cleanup in backend

disabled

check if backend has disabled submission

kick_backend

kick backend after a request's priority has changed

bump_inflight_request_prio

update priority of an inflight request

retire_inflight_request_prio

indicate request is retired to priority tracking

schedule

adjust priority of request

Call when the priority on a request has changed and it and its dependencies may need rescheduling. Note the request itself may not be ready to run!

Description

A schedule engine represents a submission queue with different priority bands. It contains all the common state (relative to the backend) to queue, track, and submit a request.

This object at the moment is quite i915 specific but will transition into a container for the `drm_gpu_scheduler` plus a few other variables once the i915 is integrated with the DRM scheduler.

Logical Rings, Logical Ring Contexts and Execlists

Motivation: GEN8 brings an expansion of the HW contexts: “Logical Ring Contexts”. These expanded contexts enable a number of new abilities, especially “Execlists” (also implemented in this file).

One of the main differences with the legacy HW contexts is that logical ring contexts incorporate many more things to the context’s state, like PDPs or ringbuffer control registers:

The reason why PDPs are included in the context is straightforward: as PP GTTs (per-process GTTs) are actually per-context, having the PDPs contained there mean you don’t need to do a `ppgtt->switch_mm` yourself, instead, the GPU will do it for you on the context switch.

But, what about the ringbuffer control registers (head, tail, etc.)? shouldn’t we just need a set of those per engine command streamer? This is where the name “Logical Rings” starts to make sense: by virtualizing the rings, the engine cs shifts to a new “ring buffer” with every context switch. When you want to submit a workload to the GPU you: A) choose your context, B) find its appropriate virtualized ring, C) write commands to it and then, finally, D) tell the GPU to switch to that context.

Instead of the legacy `MI_SET_CONTEXT`, the way you tell the GPU to switch to a contexts is via a context execution list, ergo “Execlists”.

LRC implementation: Regarding the creation of contexts, we have:

- One global default context.
- One local default context for each opened fd.
- One local extra context for each context create ioctl call.

Now that ringbuffers belong per-context (and not per-engine, like before) and that contexts are uniquely tied to a given engine (and not reusable, like before) we need:

- One ringbuffer per-engine inside each context.
- One backing object per-engine inside each context.

The global default context starts its life with these new objects fully allocated and populated. The local default context for each opened fd is more complex, because we don’t know at creation time which engine is going to use them. To handle this, we have implemented a deferred creation of LR contexts:

The local context starts its life as a hollow or blank holder, that only gets populated for a given engine once we receive an execbuffer. If later on we receive another execbuffer ioctl for the same context but a different engine, we allocate/populate a new ringbuffer and context backing object and so on.

Finally, regarding local contexts created using the ioctl call: as they are only allowed with the render ring, we can allocate & populate them right away (no need to defer anything, at least for now).

Execlists implementation: Execlists are the new method by which, on gen8+ hardware, workloads are submitted for execution (as opposed to the legacy, ringbuffer-based, method). This method works as follows:

When a request is committed, its commands (the BB start and any leading or trailing commands, like the seqno breadcrumbs) are placed in the ringbuffer for the appropriate context. The tail pointer in the hardware context is not updated at this time, but instead, kept by the driver in

the ringbuffer structure. A structure representing this request is added to a request queue for the appropriate engine: this structure contains a copy of the context's tail after the request was written to the ring buffer and a pointer to the context itself.

If the engine's request queue was empty before the request was added, the queue is processed immediately. Otherwise the queue will be processed during a context switch interrupt. In any case, elements on the queue will get sent (in pairs) to the GPU's ExecLists Submit Port (ELSP, for short) with a globally unique 20-bits submission ID.

When execution of a request completes, the GPU updates the context status buffer with a context complete event and generates a context switch interrupt. During the interrupt handling, the driver examines the events in the buffer: for each context complete event, if the announced ID matches that on the head of the request queue, then that request is retired and removed from the queue.

After processing, if any requests were retired and the queue is not empty then a new execution list can be submitted. The two requests at the front of the queue are next to be submitted but since a context may not occur twice in an execution list, if subsequent requests have the same ID as the first then the two requests must be combined. This is done simply by discarding requests at the head of the queue until either only one request is left (in which case we use a NULL second context) or the first two requests have unique IDs.

By always executing the first two requests in the queue the driver ensures that the GPU is kept as busy as possible. In the case where a single context completes but a second context is still executing, the request for this second context will be at the head of the queue when we remove the first one. This request will then be resubmitted along with a new request for a different context, which will cause the hardware to continue executing the second request and queue the new request (the GPU detects the condition of a context getting preempted with the same context and optimizes the context switch flow by not doing preemption, but just sampling the new tail pointer).

Global GTT views

Background and previous state

Historically objects could exist (be bound) in global GTT space only as singular instances with a view representing all of the object's backing pages in a linear fashion. This view will be called a normal view.

To support multiple views of the same object, where the number of mapped pages is not equal to the backing store, or where the layout of the pages is not linear, concept of a GGTT view was added.

One example of an alternative view is a stereo display driven by a single image. In this case we would have a framebuffer looking like this (2x2 pages):

12 34

Above would represent a normal GGTT view as normally mapped for GPU or CPU rendering. In contrast, fed to the display engine would be an alternative view which could look something like this:

1212 3434

In this example both the size and layout of pages in the alternative view is different from the normal view.

Implementation and usage

GGTT views are implemented using VMAs and are distinguished via enum `i915_gtt_view_type` and struct `i915_gtt_view`.

A new flavour of core GEM functions which work with GGTT bound objects were added with the `_ggtt_infix`, and sometimes with `_view` postfix to avoid renaming in large amounts of code. They take the struct `i915_gtt_view` parameter encapsulating all metadata required to implement a view.

As a helper for callers which are only interested in the normal view, globally const `i915_gtt_view_normal` singleton instance exists. All old core GEM API functions, the ones not taking the view parameter, are operating on, or with the normal GGTT view.

Code wanting to add or use a new GGTT view needs to:

1. Add a new enum with a suitable name.
2. Extend the metadata in the `i915_gtt_view` structure if required.
3. Add support to `i915_get_vma_pages()`.

New views are required to build a scatter-gather table from within the `i915_get_vma_pages` function. This table is stored in the `vma.gtt_view` and exists for the lifetime of an VMA.

Core API is designed to have copy semantics which means that passed in struct `i915_gtt_view` does not need to be persistent (left around after calling the core API functions).

```
int i915_gem_gtt_reserve(struct i915_address_space *vm, struct i915_gem_ww_ctx *ww,
                        struct drm_mm_node *node, u64 size, u64 offset, unsigned long
                        color, unsigned int flags)
```

reserve a node in an address_space (GTT)

Parameters

struct i915_address_space *vm
the struct `i915_address_space`

struct i915_gem_ww_ctx *ww
An optional struct `i915_gem_ww_ctx`.

struct drm_mm_node *node
the *struct `drm_mm_node`* (typically `i915_vma.mode`)

u64 size
how much space to allocate inside the GTT, must be `#I915_GTT_PAGE_SIZE` aligned

u64 offset
where to insert inside the GTT, must be `#I915_GTT_MIN_ALIGNMENT` aligned, and the node (**offset** + **size**) must fit within the address space

unsigned long color
color to apply to node, if this node is not from a VMA, color must be `#I915_COLOR_UNEVICTABLE`

unsigned int flags
control search and eviction behaviour

Description

i915_gem_gtt_reserve() tries to insert the **node** at the exact **offset** inside the address space (using **size** and **color**). If the **node** does not fit, it tries to evict any overlapping nodes from the GTT, including any neighbouring nodes if the colors do not match (to ensure guard pages between differing domains). See *i915_gem_evict_for_node()* for the gory details on the eviction algorithm. `#PIN_NONBLOCK` may used to prevent waiting on evicting active overlapping objects, and any overlapping node that is pinned or marked as unevictable will also result in failure.

Return

0 on success, `-ENOSPC` if no suitable hole is found, `-EINTR` if asked to wait for eviction and interrupted.

```
int i915_gem_gtt_insert(struct i915_address_space *vm, struct i915_gem_ww_ctx *ww,
                      struct drm_mm_node *node, u64 size, u64 alignment, unsigned
                      long color, u64 start, u64 end, unsigned int flags)
```

insert a node into an address_space (GTT)

Parameters

struct i915_address_space *vm
the struct `i915_address_space`

struct i915_gem_ww_ctx *ww
An optional struct `i915_gem_ww_ctx`.

struct drm_mm_node *node
the *struct `drm_mm_node`* (typically `i915_vma.node`)

u64 size
how much space to allocate inside the GTT, must be `#I915_GTT_PAGE_SIZE` aligned

u64 alignment
required alignment of starting offset, may be 0 but if specified, this must be a power-of-two and at least `#I915_GTT_MIN_ALIGNMENT`

unsigned long color
color to apply to node

u64 start
start of any range restriction inside GTT (0 for all), must be `#I915_GTT_PAGE_SIZE` aligned

u64 end
end of any range restriction inside GTT (`U64_MAX` for all), must be `#I915_GTT_PAGE_SIZE` aligned if not `U64_MAX`

unsigned int flags
control search and eviction behaviour

Description

i915_gem_gtt_insert() first searches for an available hole into which it can insert the node. The hole address is aligned to **alignment** and its **size** must then fit entirely within the [**start**, **end**] bounds. The nodes on either side of the hole must match **color**, or else a guard page will be inserted between the two nodes (or the node evicted). If no suitable hole is found, first a victim is randomly selected and tested for eviction, otherwise then the LRU list of objects within the GTT is scanned to find the first set of replacement nodes to create the hole. Those old overlapping nodes are evicted from the GTT (and so must be rebound before any future use).

Any node that is currently pinned cannot be evicted (see `i915_vma_pin()`). Similar if the node's VMA is currently active and `#PIN_NONBLOCK` is specified, that node is also skipped when searching for an eviction candidate. See [*i915_gem_evict_something\(\)*](#) for the gory details on the eviction algorithm.

Return

0 on success, `-ENOSPC` if no suitable hole is found, `-EINTR` if asked to wait for eviction and interrupted.

GTT Fences and Swizzling

void **i915_vma_revoke_fence**(struct i915_vma *vma)
force-remove fence for a VMA

Parameters

struct i915_vma *vma
vma to map linearly (not through a fence reg)

Description

This function force-removes any fence from the given object, which is useful if the kernel wants to do untiled GTT access.

int **i915_vma_pin_fence**(struct i915_vma *vma)
set up fencing for a vma

Parameters

struct i915_vma *vma
vma to map through a fence reg

Description

When mapping objects through the GTT, userspace wants to be able to write to them without having to worry about swizzling if the object is tiled. This function walks the fence regs looking for a free one for **obj**, stealing one if it can't find any.

It then sets up the reg based on the object's properties: address, pitch and tiling format.

For an untiled surface, this removes any existing fence.

0 on success, negative error code on failure.

Return

struct i915_fence_reg ***i915_reserve_fence**(struct i915_ggtt *ggtt)
Reserve a fence for vGPU

Parameters

struct i915_ggtt *ggtt
Global GTT

Description

This function walks the fence regs looking for a free one and remove it from the `fence_list`. It is used to reserve fence for vGPU to use.

void **i915_unreserve_fence**(struct i915_fence_reg *fence)

Reclaim a reserved fence

Parameters

struct i915_fence_reg *fence

the fence reg

Description

This function add a reserved fence register from vGPU to the fence_list.

void **intel_gggtt_restore_fences**(struct i915_gggtt *gggtt)

restore fence state

Parameters

struct i915_gggtt *gggtt

Global GTT

Description

Restore the hw fence state to match the software tracking again, to be called after a gpu reset and on resume. Note that on runtime suspend we only cancel the fences, to be reacquired by the user later.

void **detect_bit_6_swizzle**(struct i915_gggtt *gggtt)

detect bit 6 swizzling pattern

Parameters

struct i915_gggtt *gggtt

Global GGTT

Description

Detects bit 6 swizzling of address lookup between IGD access and CPU access through main memory.

void **i915_gem_object_do_bit_17_swizzle**(struct drm_i915_gem_object *obj, struct sg_table *pages)

fixup bit 17 swizzling

Parameters

struct drm_i915_gem_object *obj

i915 GEM buffer object

struct sg_table *pages

the scattergather list of physical pages

Description

This function fixes up the swizzling in case any page frame number for this object has changed in bit 17 since that state has been saved with [*i915_gem_object_save_bit_17_swizzle\(\)*](#).

This is called when pinning backing storage again, since the kernel is free to move unpinned backing storage around (either by directly moving pages or by swapping them out and back in again).


```
void i915_gem_object_save_bit_17_swizzle(struct drm_i915_gem_object *obj, struct
                                         sg_table *pages)
```

save bit 17 swizzling

Parameters

```
struct drm_i915_gem_object *obj
```

i915 GEM buffer object

```
struct sg_table *pages
```

the scattergather list of physical pages

Description

This function saves the bit 17 of each page frame number so that swizzling can be fixed up later on with [i915_gem_object_do_bit_17_swizzle\(\)](#). This must be called before the backing storage can be unpinned.

Global GTT Fence Handling

Important to avoid confusions: “fences” in the i915 driver are not execution fences used to track command completion but hardware detiler objects which wrap a given range of the global GTT. Each platform has only a fairly limited set of these objects.

Fences are used to detile GTT memory mappings. They’re also connected to the hardware frontbuffer render tracking and hence interact with frontbuffer compression. Furthermore on older platforms fences are required for tiled objects used by the display engine. They can also be used by the render engine - they’re required for blitter commands and are optional for render commands. But on gen4+ both display (with the exception of fbc) and rendering have their own tiling state bits and don’t need fences.

Also note that fences only support X and Y tiling and hence can’t be used for the fancier new tiling formats like W, Ys and Yf.

Finally note that because fences are such a restricted resource they’re dynamically associated with objects. Furthermore fence state is committed to the hardware lazily to avoid unnecessary stalls on gen2/3. Therefore code must explicitly call `i915_gem_object_get_fence()` to synchronize fencing status for cpu access. Also note that some code wants an unfenced view, for those cases the fence can be removed forcefully with `i915_gem_object_put_fence()`.

Internally these functions will synchronize with userspace access by removing CPU ptes into GTT mmap (not the GTT ptes themselves) as needed.

Hardware Tiling and Swizzling Details

The idea behind tiling is to increase cache hit rates by rearranging pixel data so that a group of pixel accesses are in the same cacheline. Performance improvement from doing this on the back/depth buffer are on the order of 30%.

Intel architectures make this somewhat more complicated, though, by adjustments made to addressing of data when the memory is in interleaved mode (matched pairs of DIMMS) to improve memory bandwidth. For interleaved memory, the CPU sends every sequential 64 bytes to an alternate memory channel so it can get the bandwidth from both.

The GPU also rearranges its accesses for increased bandwidth to interleaved memory, and it matches what the CPU does for non-tiled. However, when tiled it does it a little differently, since one walks addresses not just in the X direction but also Y. So, along with alternating channels when bit 6 of the address flips, it also alternates when other bits flip -- Bits 9 (every 512 bytes, an X tile scanline) and 10 (every two X tile scanlines) are common to both the 915 and 965-class hardware.

The CPU also sometimes XORs in higher bits as well, to improve bandwidth doing strided access like we do so frequently in graphics. This is called “Channel XOR Randomization” in the MCH documentation. The result is that the CPU is XORing in either bit 11 or bit 17 to bit 6 of its address decode.

All of this bit 6 XORing has an effect on our memory management, as we need to make sure that the 3d driver can correctly address object contents.

If we don't have interleaved memory, all tiling is safe and no swizzling is required.

When bit 17 is XORed in, we simply refuse to tile at all. Bit 17 is not just a page offset, so as we page an object out and back in, individual pages in it will have different bit 17 addresses, resulting in each 64 bytes being swapped with its neighbor!

Otherwise, if interleaved, we have to tell the 3d driver what the address swizzling it needs to do is, since it's writing with the CPU to the pages (bit 6 and potentially bit 11 XORed in), and the GPU is reading from the pages (bit 6, 9, and 10 XORed in), resulting in a cumulative bit swizzling required by the CPU of XORing in bit 6, 9, 10, and potentially 11, in order to match what the GPU expects.

Object Tiling IOCTLs

```
u32 i915_gem_fence_size(struct drm_i915_private *i915, u32 size, unsigned int tiling,
                          unsigned int stride)
    required global GTT size for a fence
```

Parameters

```
struct drm_i915_private *i915
    i915 device
```

u32 size
object size

```
unsigned int tiling
    tiling mode
```

```
unsigned int stride
    tiling stride
```

Description

Return the required global GTT size for a fence (view of a tiled object), taking into account potential fence register mapping.

```
u32 i915_gem_fence_alignment(struct drm_i915_private *i915, u32 size, unsigned int tiling,
                               unsigned int stride)
    required global GTT alignment for a fence
```

Parameters

struct drm_i915_private *i915
i915 device

u32 size
object size

unsigned int tiling
tiling mode

unsigned int stride
tiling stride

Description

Return the required global GTT alignment for a fence (a view of a tiled object), taking into account potential fence register mapping.

int **i915_gem_set_tiling_ioctl**(struct *drm_device* *dev, void *data, struct *drm_file* *file)
IOCTL handler to set tiling mode

Parameters

struct drm_device *dev
DRM device

void *data
data pointer for the ioctl

struct drm_file *file
DRM file for the ioctl call

Description

Sets the tiling mode of an object, returning the required swizzling of bit 6 of addresses in the object.

Called by the user via ioctl.

Return

Zero on success, negative errno on failure.

int **i915_gem_get_tiling_ioctl**(struct *drm_device* *dev, void *data, struct *drm_file* *file)
IOCTL handler to get tiling mode

Parameters

struct drm_device *dev
DRM device

void *data
data pointer for the ioctl

struct drm_file *file
DRM file for the ioctl call

Description

Returns the current tiling mode and required bit 6 swizzling for the object.

Called by the user via ioctl.

Return

Zero on success, negative errno on failure.

`i915_gem_set_tiling_ioctl()` and `i915_gem_get_tiling_ioctl()` is the userspace interface to declare fence register requirements.

In principle GEM doesn't care at all about the internal data layout of an object, and hence it also doesn't care about tiling or swizzling. There's two exceptions:

- For X and Y tiling the hardware provides detilers for CPU access, so called fences. Since there's only a limited amount of them the kernel must manage these, and therefore userspace must tell the kernel the object tiling if it wants to use fences for detiling.
- On gen3 and gen4 platforms have a swizzling pattern for tiled objects which depends upon the physical page frame number. When swapping such objects the page frame number might change and the kernel must be able to fix this up and hence now the tiling. Note that on a subset of platforms with asymmetric memory channel population the swizzling pattern changes in an unknown way, and for those the kernel simply forbids swapping completely.

Since neither of this applies for new tiling layouts on modern platforms like W, Ys and Yf tiling GEM only allows object tiling to be set to X or Y tiled. Anything else can be handled in userspace entirely without the kernel's involvement.

Protected Objects

PXP (Protected Xe Path) is a feature available in Gen12 and newer platforms. It allows execution and flip to display of protected (i.e. encrypted) objects. The SW support is enabled via the `CONFIG_DRM_I915_PXP` kconfig.

Objects can opt-in to PXP encryption at creation time via the `I915_GEM_CREATE_EXT_PROTECTED_CONTENT` `create_ext` flag. For objects to be correctly protected they must be used in conjunction with a context created with the `I915_CONTEXT_PARAM_PROTECTED_CONTENT` flag. See the documentation of those two uapi flags for details and restrictions.

Protected objects are tied to a pxp session; currently we only support one session, which i915 manages and whose index is available in the uapi (`I915_PROTECTED_CONTENT_DEFAULT_SESSION`) for use in instructions targeting protected objects. The session is invalidated by the HW when certain events occur (e.g. suspend/resume). When this happens, all the objects that were used with the session are marked as invalid and all contexts marked as using protected content are banned. Any further attempt at using them in an `execbuf` call is rejected, while flips are converted to black frames.

Some of the PXP setup operations are performed by the Management Engine, which is handled by the mei driver; communication between i915 and mei is performed via the `mei_pxp` component module.

struct **intel_pxp**
 pxp state

Definition:

```
struct intel_pxp {
    struct intel_gt *ctrl_gt;
    u32 kcr_base;
```

```

struct gscs_session_resources {
    u64 host_session_handle;
    struct intel_context *ce;
    struct i915_vma *pkt_vma;
    void *pkt_vaddr;
    struct i915_vma *bb_vma;
    void *bb_vaddr;
} gscs_res;
struct i915_pxp_component *pxp_component;
struct device_link *dev_link;
bool pxp_component_added;
struct intel_context *ce;
struct mutex arb_mutex;
bool arb_is_valid;
u32 key_instance;
struct mutex tee_mutex;
struct {
    struct drm_i915_gem_object *obj;
    void *vaddr;
} stream_cmd;
bool hw_state_invalidated;
bool irq_enabled;
struct completion termination;
struct work_struct session_work;
u32 session_events;
#define PXP_TERMINATION_REQUEST BIT(0);
#define PXP_TERMINATION_COMPLETE BIT(1);
#define PXP_INVALID_REQUIRED BIT(2);
};

```

Members

ctrl_gt

pointer to the tile that owns the controls for PXP subsystem assets that the VDBOX, the KCR engine (and GSC CS depending on the platform)

kcr_base

base mmio offset for the KCR engine which is different on legacy platforms vs newer platforms where the KCR is inside the media-tile.

gscs_res

resources for request submission for platforms that have a GSC engine.

pxp_component

i915_pxp_component struct of the bound mei_pxp module. Only set and cleared inside component bind/unbind functions, which are protected by tee_mutex.

dev_link

Enforce module relationship for power management ordering.

pxp_component_added

track if the pxp component has been added. Set and cleared in tee init and fini functions respectively.

ce

kernel-owned context used for PXP operations

arb_mutex

protects arb session start

arb_is_valid

tracks arb session status. After a teardown, the arb session can still be in play on the HW even if the keys are gone, so we can't rely on the HW state of the session to know if it's valid and need to track the status in SW.

key_instance

tracks which key instance we're on, so we can use it to determine if an object was created using the current key or a previous one.

tee_mutex

protects the tee channel binding and messaging.

stream_cmd

LMEM obj used to send stream PXP commands to the GSC

hw_state_invalidated

if the HW perceives an attack on the integrity of the encryption it will invalidate the keys and expect SW to re-initialize the session. We keep track of this state to make sure we only re-start the arb session when required.

irq_enabled

tracks the status of the kcr irqs

termination

tracks the status of a pending termination. Only re-initialized under `gt->irq_lock` and completed in `session_work`.

session_work

worker that manages session events.

session_events

pending session events, protected with `gt->irq_lock`.

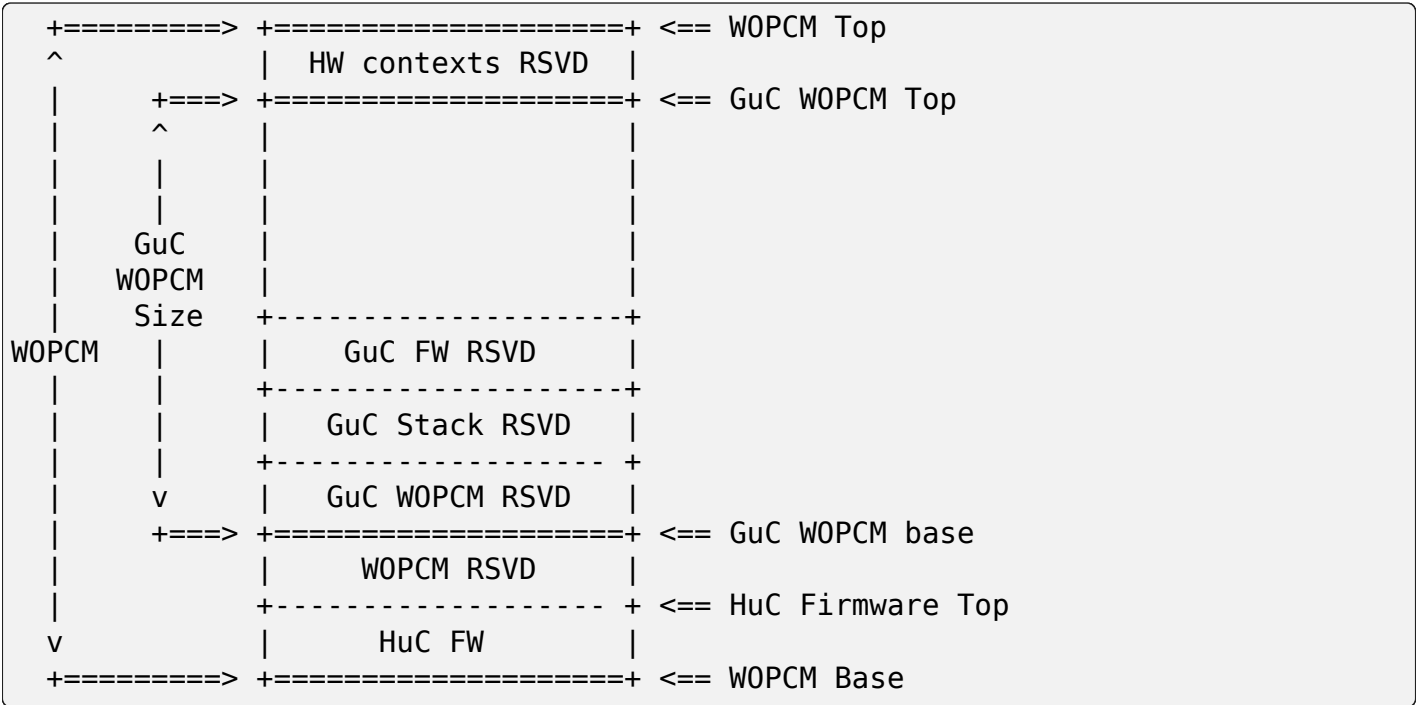
10.2.5 Microcontrollers

Starting from gen9, three microcontrollers are available on the HW: the graphics microcontroller (GuC), the HEVC/H.265 microcontroller (HuC) and the display microcontroller (DMC). The driver is responsible for loading the firmwares on the microcontrollers; the GuC and HuC firmwares are transferred to WOPCM using the DMA engine, while the DMC firmware is written through MMIO.

WOPCM

WOPCM Layout

The layout of the WOPCM will be fixed after writing to GuC WOPCM size and offset registers whose values are calculated and determined by HuC/GuC firmware size and set of hardware requirements/restrictions as shown below:



GuC accessible WOPCM starts at GuC WOPCM base and ends at GuC WOPCM top. The top part of the WOPCM is reserved for hardware contexts (e.g. RC6 context).

GuC

The GuC is a microcontroller inside the GT HW, introduced in gen9. The GuC is designed to offload some of the functionality usually performed by the host driver; currently the main operations it can take care of are:

- Authentication of the HuC, which is required to fully enable HuC usage.
- Low latency graphics context scheduling (a.k.a. GuC submission).
- GT Power management.

The `enable_guc` module parameter can be used to select which of those operations to enable within GuC. Note that not all the operations are supported on all gen9+ platforms.

Enabling the GuC is not mandatory and therefore the firmware is only loaded if at least one of the operations is selected. However, not loading the GuC might result in the loss of some features that do require the GuC (currently just the HuC, but more are expected to land in the future).

struct **intel_guc**

Top level structure of GuC.

Definition:

```
struct intel_guc {
    struct intel_uc_fw fw;
    struct intel_guc_log log;
    struct intel_guc_ct ct;
    struct intel_guc_slpc slpc;
    struct intel_guc_state_capture *capture;
    struct dentry *dbgfs_node;
    struct i915_sched_engine *sched_engine;
    struct i915_request *stalled_request;
    enum {
        STALL_NONE,
        STALL_REGISTER_CONTEXT,
        STALL_MOVE_LRC_TAIL,
        STALL_ADD_REQUEST,
    } submission_stall_reason;
    spinlock_t irq_lock;
    unsigned int msg_enabled_mask;
    atomic_t outstanding_submission_g2h;
    struct {
        bool enabled;
        void (*reset)(struct intel_guc *guc);
        void (*enable)(struct intel_guc *guc);
        void (*disable)(struct intel_guc *guc);
    } interrupts;
    struct {
        spinlock_t lock;
        struct ida guc_ids;
        int num_guc_ids;
        unsigned long *guc_ids_bitmap;
        struct list_head guc_id_list;
        unsigned int guc_ids_in_use;
        struct list_head destroyed_contexts;
        struct work_struct destroyed_worker;
        struct work_struct reset_fail_worker;
        intel_engine_mask_t reset_fail_mask;
        unsigned int sched_disable_delay_ms;
        unsigned int sched_disable_gucid_threshold;
    } submission_state;
    bool submission_supported;
    bool submission_selected;
```



```

bool submission_initialized;
struct intel_uc_fw_ver submission_version;
bool rc_supported;
bool rc_selected;
struct i915_vma *ads_vma;
struct iosys_map ads_map;
u32 ads_regset_size;
u32 ads_regset_count[I915_NUM_ENGINES];
struct guc_mmio_reg *ads_regset;
u32 ads_golden_ctxt_size;
u32 ads_capture_size;
u32 ads_engine_usage_size;
struct i915_vma *lrc_desc_pool_v69;
void *lrc_desc_pool_vaddr_v69;
struct xarray context_lookup;
u32 params[GUC_CTL_MAX_DWORDS];
struct {
    u32 base;
    unsigned int count;
    enum forcewake_domains fw_domains;
} send_regs;
i915_reg_t notify_reg;
u32 mmio_msg;
struct mutex send_mutex;
struct {
    spinlock_t lock;
    u64 gt_stamp;
    unsigned long ping_delay;
    struct delayed_work work;
    u32 shift;
    unsigned long last_stat_jiffies;
} timestamp;
#ifdef CONFIG_DRM_I915_SELFTEST;
    int number_guc_id_stolen;
#endif;
};

```

Members

fw

the GuC firmware

log

sub-structure containing GuC log related data and objects

ct

the command transport communication channel

slpc

sub-structure containing SLPC related data and objects

capture

the error-state-capture module's data and objects

dbgfs_node

debugfs node

sched_engine

Global engine used to submit requests to GuC

stalled_request

if GuC can't process a request for any reason, we save it until GuC restarts processing. No other request can be submitted until the stalled request is processed.

submission_stall_reason

reason why submission is stalled

irq_lock

protects GuC irq state

msg_enabled_mask

mask of events that are processed when receiving an INTEL_GUC_ACTION_DEFAULT G2H message.

outstanding_submission_g2h

number of outstanding GuC to Host responses related to GuC submission, used to determine if the GT is idle

interrupts

pointers to GuC interrupt-managing functions.

submission_state

sub-structure for submission state protected by single lock

submission_supported

tracks whether we support GuC submission on the current platform

submission_selected

tracks whether the user enabled GuC submission

submission_initialized

tracks whether GuC submission has been initialised

submission_version

Submission API version of the currently loaded firmware

rc_supported

tracks whether we support GuC rc on the current platform

rc_selected

tracks whether the user enabled GuC rc

ads_vma

object allocated to hold the GuC ADS

ads_map

contents of the GuC ADS

ads_regset_size

size of the save/restore regsets in the ADS

ads_regset_count

number of save/restore registers in the ADS for each engine

ads_regset

save/restore regsets in the ADS

ads_golden_ctxt_size

size of the golden contexts in the ADS

ads_capture_size

size of register lists in the ADS used for error capture

ads_engine_usage_size

size of engine usage in the ADS

lrc_desc_pool_v69

object allocated to hold the GuC LRC descriptor pool

lrc_desc_pool_vaddr_v69

contents of the GuC LRC descriptor pool

context_lookup

used to resolve intel_context from guc_id, if a context is present in this structure it is registered with the GuC

params

Control params for fw initialization

send_regs

GuC's FW specific registers used for sending MMIO H2G

notify_reg

register used to send interrupts to the GuC FW

mmio_msg

notification bitmask that the GuC writes in one of its registers when the CT channel is disabled, to be processed when the channel is back up.

send_mutex

used to serialize the intel_guc_send actions

timestamp

GT timestamp object that stores a copy of the timestamp and adjusts it for overflow using a worker.

number_guc_id_stolen

The number of guc_ids that have been stolen

Description

It handles firmware loading and manages client pool. intel_guc owns an i915_sched_engine for submission.

u32 **intel_guc_ggtt_offset**(struct *intel_guc* *guc, struct i915_vma *vma)

Get and validate the GGTT offset of **vma**

Parameters

struct intel_guc *guc

intel_guc structure.

struct i915_vma *vma

i915 graphics virtual memory area.

Description

GuC does not allow any gfx GGTT address that falls into range [0, ggtt.pin_bias), which is reserved for Boot ROM, SRAM and WOPCM. Currently, in order to exclude [0, ggtt.pin_bias) address space from GGTT, all gfx objects used by GuC are allocated with `intel_guc_allocate_vma()` and pinned with PIN_OFFSET_BIAS along with the value of ggtt.pin_bias.

Return

GGTT offset of the **vma**.

GuC Firmware Layout

The GuC/HuC firmware layout looks like this:

```

+=====+
|  Firmware blob                               |
+=====+
|  CSS header  |      uCode      |  RSA key  |  modulus  |  exponent  |
+=====+
<-header size->          <---header size continued ----->
<--- size ----->
                                <-key size->
                                    <-mod size->
                                        <-exp size->

```

The firmware may or may not have modulus key and exponent data. The header, uCode and RSA signature are must-have components that will be used by driver. Length of each components, which is all in dwords, can be found in header. In the case that modulus and exponent are not present in fw, a.k.a truncated image, the length value still appears in header.

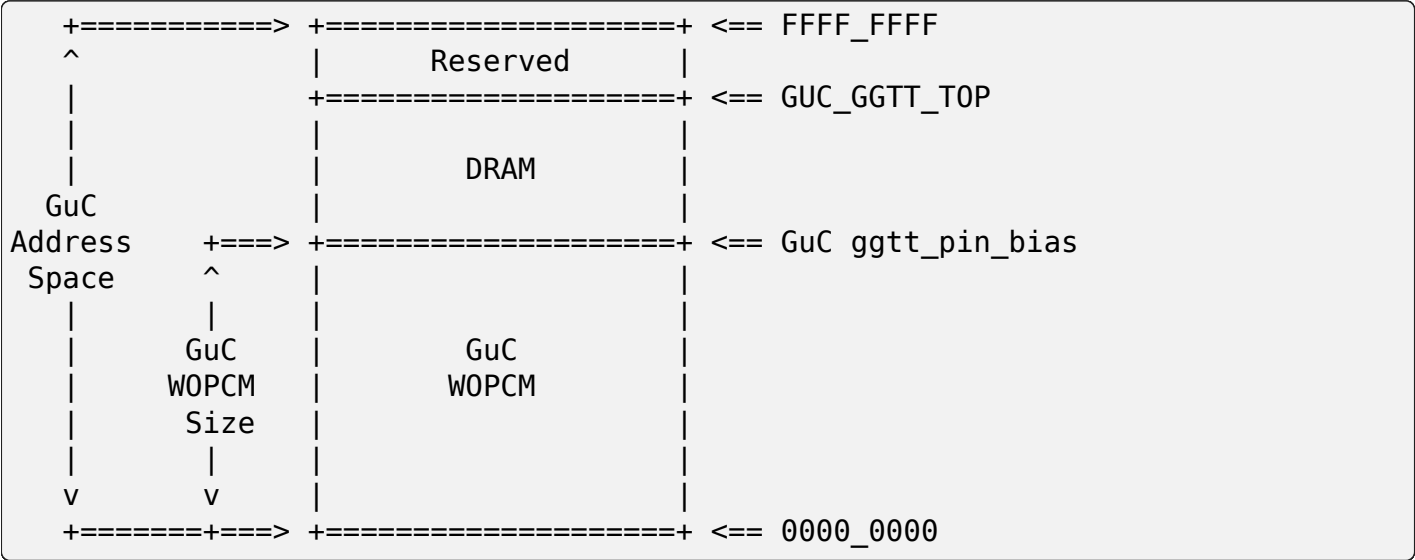
Driver will do some basic fw size validation based on the following rules:

1. Header, uCode and RSA are must-have components.
2. All firmware components, if they present, are in the sequence illustrated in the layout table above.
3. Length info of each component can be found in header, in dwords.
4. Modulus and exponent key are not required by driver. They may not appear in fw. So driver will load a truncated firmware in this case.

Starting from DG2, the HuC is loaded by the GSC instead of i915. The GSC firmware performs all the required integrity checks, we just need to check the version. Note that the header for GSC-managed blobs is different from the CSS used for dma-loaded firmwares.

GuC Memory Management

GuC can't allocate any memory for its own usage, so all the allocations must be handled by the host driver. GuC accesses the memory via the GGTT, with the exception of the top and bottom parts of the 4GB address space, which are instead re-mapped by the GuC HW to memory location of the FW itself (WOPCM) or other parts of the HW. The driver must take care not to place objects that the GuC is going to access in these reserved ranges. The layout of the GuC address space is shown below:



The lower part of GuC Address Space [0, gggtt_pin_bias) is mapped to GuC WOPCM while upper part of GuC Address Space [gggtt_pin_bias, GUC_GGTT_TOP) is mapped to DRAM. The value of the GuC gggtt_pin_bias is the GuC WOPCM size.

```
struct i915_vma *intel_guc_allocate_vma(struct intel_guc *guc, u32 size)
    Allocate a GGTT VMA for GuC usage
```

Parameters

struct intel_guc *guc
the guc

u32 size
size of area to allocate (both virtual space and memory)

Description

This is a wrapper to create an object for use with the GuC. In order to use it inside the GuC, an object needs to be pinned lifetime, so we allocate both some backing storage and a range inside the Global GTT. We must pin it in the GGTT somewhere other than than [0, GUC gggtt_pin_bias) because that range is reserved inside GuC.

Return

A i915_vma if successful, otherwise an ERR_PTR.

GuC-specific firmware loader

int **intel_guc_fw_upload**(struct *intel_guc* *guc)
load GuC uCode to device

Parameters

struct **intel_guc** *guc
intel_guc structure

Description

Called from `intel_uc_init_hw()` during driver load, resume from sleep and after a GPU reset.

The firmware image should have already been fetched into memory, so only check that fetch succeeded, and then transfer the image to the h/w.

Return

non-zero code on error

GuC-based command submission

The Scratch registers: There are 16 MMIO-based registers start from 0xC180. The kernel driver writes a value to the action register (SOFT_SCRATCH_0) along with any data. It then triggers an interrupt on the GuC via another register write (0xC4C8). Firmware writes a success/fail code back to the action register after processes the request. The kernel driver polls waiting for this update and then proceeds.

Command Transport buffers (CTBs): Covered in detail in other sections but CTBs (Host to GuC - H2G, GuC to Host - G2H) are a message interface between the i915 and GuC.

Context registration: Before a context can be submitted it must be registered with the GuC via a H2G. A unique `guc_id` is associated with each context. The context is either registered at request creation time (normal operation) or at submission time (abnormal operation, e.g. after a reset).

Context submission: The i915 updates the LRC tail value in memory. The i915 must enable the scheduling of the context within the GuC for the GuC to actually consider it. Therefore, the first time a disabled context is submitted we use a schedule enable H2G, while follow up submissions are done via the context submit H2G, which informs the GuC that a previously enabled context has new work available.

Context unpin: To unpin a context a H2G is used to disable scheduling. When the corresponding G2H returns indicating the scheduling disable operation has completed it is safe to unpin the context. While a disable is in flight it isn't safe to resubmit the context so a fence is used to stall all future requests of that context until the G2H is returned. Because this interaction with the GuC takes a non-zero amount of time we delay the disabling of scheduling after the pin count goes to zero by a configurable period of time (see `SCHED_DISABLE_DELAY_MS`). The thought is this gives the user a window of time to resubmit something on the context before doing this costly operation. This delay is only done if the context isn't closed and the `guc_id` usage is less than a threshold (see `NUM_SCHED_DISABLE_GUC_IDS_THRESHOLD`).

Context deregistration: Before a context can be destroyed or if we steal its `guc_id` we must deregister the context with the GuC via H2G. If stealing the `guc_id` it isn't safe to submit anything to this `guc_id` until the deregister completes so a fence is used to stall all requests as-

sociated with this `guc_id` until the corresponding G2H returns indicating the `guc_id` has been deregistered.

`submission_state.guc_ids`: Unique number associated with private GuC context data passed in during context registration / submission / deregistration. 64k available. Simple ida is used for allocation.

Stealing `guc_ids`: If no `guc_ids` are available they can be stolen from another context at request creation time if that context is unpinned. If a `guc_id` can't be found we punt this problem to the user as we believe this is near impossible to hit during normal use cases.

Locking: In the GuC submission code we have 3 basic spin locks which protect everything. Details about each below.

`sched_engine->lock` This is the submission lock for all contexts that share an i915 schedule engine (`sched_engine`), thus only one of the contexts which share a `sched_engine` can be submitting at a time. Currently only one `sched_engine` is used for all of GuC submission but that could change in the future.

`guc->submission_state.lock` Global lock for GuC submission state. Protects `guc_ids` and destroyed contexts list.

`ce->guc_state.lock` Protects everything under `ce->guc_state`. Ensures that a context is in the correct state before issuing a H2G. e.g. We don't issue a schedule disable on a disabled context (bad idea), we don't issue a schedule enable when a schedule disable is in flight, etc... Also protects list of inflight requests on the context and the priority management state. Lock is individual to each context.

Lock ordering rules: `sched_engine->lock -> ce->guc_state.lock guc->submission_state.lock -> ce->guc_state.lock`

Reset races: When a full GT reset is triggered it is assumed that some G2H responses to H2Gs can be lost as the GuC is also reset. Losing these G2H can prove to be fatal as we do certain operations upon receiving a G2H (e.g. destroy contexts, release `guc_ids`, etc...). When this occurs we can scrub the context state and cleanup appropriately, however this is quite racey. To avoid races, the reset code must disable submission before scrubbing for the missing G2H, while the submission code must check for submission being disabled and skip sending H2Gs and updating context states when it is. Both sides must also make sure to hold the relevant locks.

GuC ABI

HXG Message

All messages exchanged with GuC are defined using 32 bit dwords. First dword is treated as a message header. Remaining dwords are optional.

	Bits	Description
0	31	ORIGIN - originator of the message <ul style="list-style-type: none"> • GUC_HXG_ORIGIN_HOST = 0 • GUC_HXG_ORIGIN_GUC = 1
	30:28	TYPE - message type <ul style="list-style-type: none"> • GUC_HXG_TYPE_REQUEST = 0 • GUC_HXG_TYPE_EVENT = 1 • GUC_HXG_TYPE_FAST_REQUEST = 2 • GUC_HXG_TYPE_NO_RESPONSE = 3 • GUC_HXG_TYPE_NO_RESPONSE_2 = 5 • GUC_HXG_TYPE_RESPONSE = 6 • GUC_HXG_TYPE_RESPONSE_2 = 7
	27:0	AUX - auxiliary data (depends on TYPE)
1	31:0	PAYLOAD - optional payload (depends on TYPE)
...		
n	31:0	

HXG Request

The *HXG Request* message should be used to initiate synchronous activity for which confirmation or return data is expected.

The recipient of this message shall use *HXG Response*, *HXG Failure* or *HXG Retry* message as a definite reply, and may use *HXG Busy* message as a intermediate reply.

Format of **DATA0** and all **DATAn** fields depends on the **ACTION** code.

	Bits	Description
0	31	ORIGIN
	30:28	TYPE = <i>GUC_HXG_TYPE_REQUEST</i>
	27:16	DATA0 - request data (depends on ACTION)
	15:0	ACTION - requested action code
1	31:0	DATAn - optional data (depends on ACTION)
...		
n	31:0	

HXG Fast Request

The *HXG Request* message should be used to initiate asynchronous activity for which confirmation or return data is not expected.

If confirmation is required then *HXG Request* shall be used instead.

The recipient of this message may only use *HXG Failure* message if it was unable to accept this request (like invalid data).

Format of *HXG Fast Request* message is same as *HXG Request* except **TYPE**.

	Bits	Description
0	31	ORIGIN - see <i>HXG Message</i>
	30:28	TYPE = <i>GUC_HXG_TYPE_FAST_REQUEST</i>
	27:16	DATA0 - see <i>HXG Request</i>
	15:0	ACTION - see <i>HXG Request</i>
...		DATAn - see <i>HXG Request</i>

HXG Event

The *HXG Event* message should be used to initiate asynchronous activity that does not involves immediate confirmation nor data.

Format of **DATA0** and all **DATAn** fields depends on the **ACTION** code.

	Bits	Description
0	31	ORIGIN
	30:28	TYPE = <i>GUC_HXG_TYPE_EVENT</i>
	27:16	DATA0 - event data (depends on ACTION)
	15:0	ACTION - event action code
1	31:0	DATAn - optional event data (depends on ACTION)
...		
n	31:0	

HXG Busy

The *HXG Busy* message may be used to acknowledge reception of the *HXG Request* message if the recipient expects that its processing will be longer than default timeout.

The **COUNTER** field may be used as a progress indicator.

	Bits	Description
0	31	ORIGIN
	30:28	TYPE = <i>GUC_HXG_TYPE_NO_RESPONSE_BUSY</i>
	27:0	COUNTER - progress indicator

HXG Retry

The *HXG Retry* message should be used by recipient to indicate that the *HXG Request* message was dropped and it should be resent again.

The **REASON** field may be used to provide additional information.

	Bits	Description
0	31	ORIGIN
	30:28	TYPE = <i>GUC_HXG_TYPE_NO_RESPONSE_RETRY</i>
	27:0	REASON - reason for retry GUC_HXG_RETRY_REASON_U = 0

HXG Failure

The *HXG Failure* message shall be used as a reply to the *HXG Request* message that could not be processed due to an error.

	Bits	Description
0	31	ORIGIN
	30:28	TYPE = <i>GUC_HXG_TYPE_RESPONSE_FAILURE</i>
	27:16	HINT - additional error hint
	15:0	ERROR - error/result code

HXG Response

The *HXG Response* message shall be used as a reply to the *HXG Request* message that was successfully processed without an error.

	Bits	Description
0	31	ORIGIN
	30:28	TYPE = <i>GUC_HXG_TYPE_RESPONSE_SUCCESS</i>
	27:0	DATA0 - data (depends on ACTION from <i>HXG Request</i>)
1	31:0	DATA_n - data (depends on ACTION from <i>HXG Request</i>)
...		
n	31:0	

GuC MMIO based communication

The MMIO based communication between Host and GuC relies on special hardware registers which format could be defined by the software (so called scratch registers).

Each MMIO based message, both Host to GuC (H2G) and GuC to Host (G2H) messages, which maximum length depends on number of available scratch registers, is directly written into those scratch registers.

For Gen9+, there are 16 software scratch registers 0xC180-0xC1B8, but no H2G command takes more than 4 parameters and the GuC firmware itself uses an 4-element array to store the H2G message.

For Gen11+, there are additional 4 registers 0x190240-0x19024C, which are, regardless on lower count, preferred over legacy ones.

The MMIO based communication is mainly used during driver initialization phase to setup the *CTB based communication* that will be used afterwards.

MMIO HXG Message

Format of the MMIO messages follows definitions of *HXG Message*.

	Bits	Description
0	31:0	[Embedded <i>HXG Message</i>]
...		
n	31:0	

CT Buffer

Circular buffer used to send *CTB Message*

CTB Descriptor

	Bits	Description
0	31:0	HEAD - offset (in dwords) to the last dword that was read from the <i>CT Buffer</i> . It can only be updated by the receiver.
1	31:0	TAIL - offset (in dwords) to the last dword that was written to the <i>CT Buffer</i> . It can only be updated by the sender.
2	31:0	STATUS - status of the CTB <ul style="list-style-type: none">• GUC_CTB_STATUS_NO_ERROR = 0 (normal operation)• GUC_CTB_STATUS_OVERFLOW = 1 (head/tail too large)• GUC_CTB_STATUS_UNDERFLOW = 2 (truncated message)• GUC_CTB_STATUS_MISMATCH = 4 (head/tail modified)• GUC_CTB_STATUS_UNUSED = 8 (CTB is not in use)
...		RESERVED = MBZ
15	31:0	RESERVED = MBZ

CTB Message

	Bits	Description
0	31:16	FENCE - message identifier
	15:12	FORMAT - format of the CTB message GUC_CTB_FORMAT_HXG = 0 - see <i>CTB HXG Message</i>
	11:8	RESERVED
	7:0	NUM_DWORDS - length of the CTB message (w/o header)
1	31:0	optional (depends on FORMAT)
...		
n	31:0	

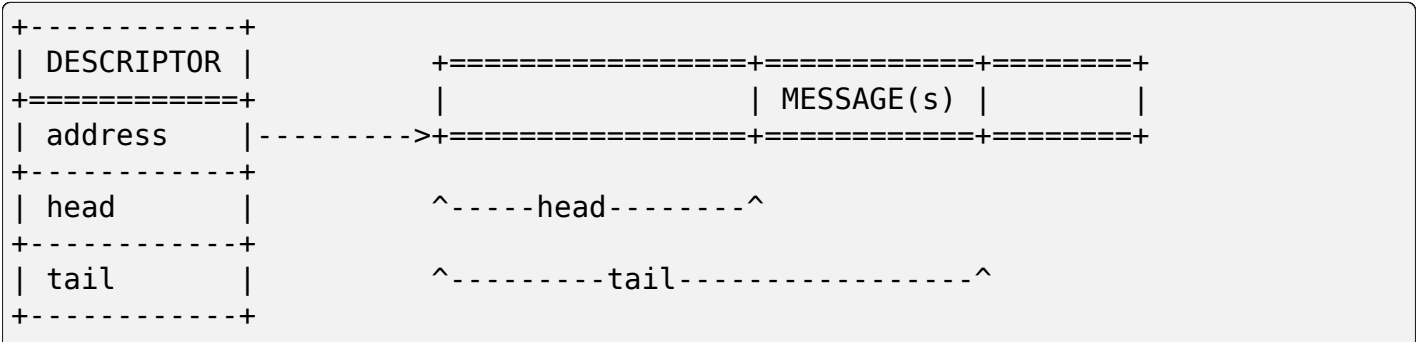
CTB HXG Message

	Bits	Description
0	31:16	FENCE
	15:12	FORMAT = <i>GUC_CTB_FORMAT_HXG</i>
	11:8	RESERVED = MBZ
	7:0	NUM_DWORDS = length (in dwords) of the embedded HXG message
1	31:0	[Embedded <i>HXG Message</i>]
...		
n	31:0	

CTB based communication

The CTB (command transport buffer) communication between Host and GuC is based on u32 data stream written to the shared buffer. One buffer can be used to transmit data only in one direction (one-directional channel).

Current status of the each buffer is stored in the buffer descriptor. Buffer descriptor holds tail and head fields that represents active data stream. The tail field is updated by the data producer (sender), and head field is updated by the data consumer (receiver):



```
| size | ^-----size-----^
+-----+
```

Each message in data stream starts with the single u32 treated as a header, followed by optional set of u32 data that makes message specific payload:

```
+-----+-----+-----+-----+
|          MESSAGE          |
+-----+-----+-----+-----+
| msg[0] | [1] | ... | [n-1] |
+-----+-----+-----+-----+
| MESSAGE | MESSAGE PAYLOAD |
+-----+-----+-----+-----+
| HEADER  | 0 | ... | n |
+-----+-----+-----+-----+
+=====+=====+=====+=====+
| 31:16| code|          |          |
+-----+-----+-----+-----+
| 15:5| flags|          |          |
+-----+-----+-----+-----+
| 4:0| len|          |          |
+-----+-----+-----+-----+
^-----len-----^
```

The message header consists of:

- **len**, indicates length of the message payload (in u32)
- **code**, indicates message code
- **flags**, holds various bits to control message handling

HOST2GUC_SELF_CFG

This message is used by Host KMD to setup of the *GuC Self Config KLVs*.

This message must be sent as *MMIO HXG Message*.

	Bits	Description
0	31	ORIGIN = GUC_HXG_ORIGIN_HOST
	30:28	TYPE = GUC_HXG_TYPE_REQUEST
	27:16	DATA0 = MBZ
	15:0	ACTION = GUC_ACTION_HOST2GUC_SELF_CFG = 0x0508
1	31:16	KLV_KEY - KLV key, see GuC Self Config KLVs
	15:0	KLV_LEN - KLV length <ul style="list-style-type: none">• 32 bit KLV = 1• 64 bit KLV = 2
2	31:0	VALUE32 - Bits 31-0 of the KLV value
3	31:0	VALUE64 - Bits 63-32 of the KLV value (KLV_LEN = 2)

	Bits	Description
0	31	ORIGIN = GUC_HXG_ORIGIN_GUC
	30:28	TYPE = GUC_HXG_TYPE_RESPONSE_SUCCESS
	27:0	DATA0 = NUM - 1 if KLV was parsed, 0 if not recognized

HOST2GUC_CONTROL_CTB

This H2G action allows Vf Host to enable or disable H2G and G2H *CT Buffer*.

This message must be sent as *MMIO HXG Message*.

	Bits	Description
0	31	ORIGIN = GUC_HXG_ORIGIN_HOST
	30:28	TYPE = GUC_HXG_TYPE_REQUEST
	27:16	DATA0 = MBZ
	15:0	ACTION = GUC_ACTION_HOST2GUC_CONTROL = 0x4509
1	31:0	CONTROL - control CTB based communication <ul style="list-style-type: none"> GUC_CTB_CONTROL_DISABLE = 0 GUC_CTB_CONTROL_ENABLE = 1

	Bits	Description
0	31	ORIGIN = GUC_HXG_ORIGIN_GUC
	30:28	TYPE = GUC_HXG_TYPE_RESPONSE_SUCCESS
	27:0	DATA0 = MBZ

GuC KLV

	Bits	Description
0	31:16	KEY - KLV key identifier <ul style="list-style-type: none"> GuC Self Config KLVs
	15:0	LEN - length of VALUE (in 32bit dwords)
1	31:0	VALUE - actual value of the KLV (format depends on KEY)
...		
n	31:0	

GuC Self Config KLVs

[GuC KLV](#) keys available for use with [HOST2GUC_SELF_CFG](#).

GUC_KLV_SELF_CFG_H2G_CTB_ADDR

[0x0902] Refers to 64 bit Global Gfx address of H2G [CT Buffer](#). Should be above WOPCM address but below APIC base address for native mode.

GUC_KLV_SELF_CFG_H2G_CTB_DESCRIPTOR_ADDR

[0x0903] Refers to 64 bit Global Gfx address of H2G [CTB Descriptor](#). Should be above

WOPCM address but below APIC base address for native mode.

GUC_KLV_SELF_CFG_H2G_CTB_SIZE

[0x0904] Refers to size of H2G *CT Buffer* in bytes. Should be a multiple of 4K.

GUC_KLV_SELF_CFG_G2H_CTB_ADDR

[0x0905] Refers to 64 bit Global Gfx address of G2H *CT Buffer*. Should be above WOPCM address but below APIC base address for native mode.

GUC_KLV_SELF_CFG_G2H_CTB_DESCRIPTOR_ADDR

[0x0906] Refers to 64 bit Global Gfx address of G2H *CTB Descriptor*. Should be above WOPCM address but below APIC base address for native mode.

GUC_KLV_SELF_CFG_G2H_CTB_SIZE

[0x0907] Refers to size of G2H *CT Buffer* in bytes. Should be a multiple of 4K.

HuC

The HuC is a dedicated microcontroller for usage in media HEVC (High Efficiency Video Coding) operations. Userspace can directly use the firmware capabilities by adding HuC specific commands to batch buffers.

The kernel driver is only responsible for loading the HuC firmware and triggering its security authentication. This is done differently depending on the platform:

- older platforms (from Gen9 to most Gen12s): the load is performed via DMA and the authentication via GuC
- DG2: load and authentication are both performed via GSC.
- MTL and newer platforms: the load is performed via DMA (same as with not-DG2 older platforms), while the authentication is done in 2-steps, a first auth for clear-media workloads via GuC and a second one for all workloads via GSC.

On platforms where the GuC does the authentication, to correctly do so the HuC binary must be loaded before the GuC one. Loading the HuC is optional; however, not using the HuC might negatively impact power usage and/or performance of media workloads, depending on the use-cases. HuC must be reloaded on events that cause the WOPCM to lose its contents (S3/S4, FLR); on older platforms the HuC must also be reloaded on GuC/GT reset, while on newer ones it will survive that.

See <https://github.com/intel/media-driver> for the latest details on HuC functionality.

int **intel_huc_auth**(struct intel_huc *huc, enum intel_huc_authentication_type type)

Authenticate HuC uCode

Parameters

struct intel_huc *huc

intel_huc structure

enum intel_huc_authentication_type type

authentication type (via GuC or via GSC)

Description

Called after HuC and GuC firmware loading during `intel_uc_init_hw()`.

This function invokes the GuC action to authenticate the HuC firmware, passing the offset of the RSA signature to `intel_guc_auth_huc()`. It then waits for up to 50ms for firmware verification ACK.

HuC Memory Management

Similarly to the GuC, the HuC can't do any memory allocations on its own, with the difference being that the allocations for HuC usage are handled by the userspace driver instead of the kernel one. The HuC accesses the memory via the PPGTT belonging to the context loaded on the VCS executing the HuC-specific commands.

HuC Firmware Layout

The HuC FW layout is the same as the GuC one, see [GuC Firmware Layout](#)

DMC

See [DMC Firmware Support](#)

10.2.6 Tracing

This sections covers all things related to the tracepoints implemented in the i915 driver.

i915_ppgtt_create and i915_ppgtt_release

With full ppgtt enabled each process using drm will allocate at least one translation table. With these traces it is possible to keep track of the allocation and of the lifetime of the tables; this can be used during testing/debug to verify that we are not leaking ppgtts. These traces identify the ppgtt through the vm pointer, which is also printed by the `i915_vma_bind` and `i915_vma_unbind` tracepoints.

i915_context_create and i915_context_free

These tracepoints are used to track creation and deletion of contexts. If full ppgtt is enabled, they also print the address of the vm assigned to the context.

10.2.7 Perf

Overview

Gen graphics supports a large number of performance counters that can help driver and application developers understand and optimize their use of the GPU.

This i915 perf interface enables userspace to configure and open a file descriptor representing a stream of GPU metrics which can then be `read()` as a stream of sample records.

The interface is particularly suited to exposing buffered metrics that are captured by DMA from the GPU, unsynchronized with and unrelated to the CPU.

Streams representing a single context are accessible to applications with a corresponding drm file descriptor, such that OpenGL can use the interface without special privileges. Access to system-wide metrics requires root privileges by default, unless changed via the `dev.i915.perf_event_paranoid` sysctl option.

Comparison with Core Perf

The interface was initially inspired by the core Perf infrastructure but some notable differences are:

i915 perf file descriptors represent a “stream” instead of an “event”; where a perf event primarily corresponds to a single 64bit value, while a stream might sample sets of tightly-coupled counters, depending on the configuration. For example the Gen OA unit isn’t designed to support orthogonal configurations of individual counters; it’s configured for a set of related counters. Samples for an i915 perf stream capturing OA metrics will include a set of counter values packed in a compact HW specific format. The OA unit supports a number of different packing formats which can be selected by the user opening the stream. Perf has support for grouping events, but each event in the group is configured, validated and authenticated individually with separate system calls.

i915 perf stream configurations are provided as an array of u64 (key,value) pairs, instead of a fixed struct with multiple miscellaneous config members, interleaved with event-type specific members.

i915 perf doesn’t support exposing metrics via an mmap’d circular buffer. The supported metrics are being written to memory by the GPU unsynchronized with the CPU, using HW specific packing formats for counter sets. Sometimes the constraints on HW configuration require reports to be filtered before it would be acceptable to expose them to unprivileged applications - to hide the metrics of other processes/contexts. For these use cases a read() based interface is a good fit, and provides an opportunity to filter data as it gets copied from the GPU mapped buffers to userspace buffers.

Issues hit with first prototype based on Core Perf

The first prototype of this driver was based on the core perf infrastructure, and while we did make that mostly work, with some changes to perf, we found we were breaking or working around too many assumptions baked into perf’s currently cpu centric design.

In the end we didn’t see a clear benefit to making perf’s implementation and interface more complex by changing design assumptions while we knew we still wouldn’t be able to use any existing perf based userspace tools.

Also considering the Gen specific nature of the Observability hardware and how userspace will sometimes need to combine i915 perf OA metrics with side-band OA data captured via `MI_REPORT_PERF_COUNT` commands; we’re expecting the interface to be used by a platform specific userspace such as OpenGL or tools. This is to say; we aren’t inherently missing out on having a standard vendor/architecture agnostic interface by not using perf.

For posterity, in case we might re-visit trying to adapt core perf to be better suited to exposing i915 metrics these were the main pain points we hit:

- The perf based OA PMU driver broke some significant design assumptions:

Existing perf pmus are used for profiling work on a cpu and we were introducing the idea of `_IS_DEVICE` pmus with different security implications, the need to fake cpu-related data (such as user/kernel registers) to fit with perf's current design, and adding `_DEVICE` records as a way to forward device-specific status records.

The OA unit writes reports of counters into a circular buffer, without involvement from the CPU, making our PMU driver the first of a kind.

Given the way we were periodically forward data from the GPU-mapped, OA buffer to perf's buffer, those bursts of sample writes looked to perf like we were sampling too fast and so we had to subvert its throttling checks.

Perf supports groups of counters and allows those to be read via transactions internally but transactions currently seem designed to be explicitly initiated from the cpu (say in response to a userspace `read()`) and while we could pull a report out of the OA buffer we can't trigger a report from the cpu on demand.

Related to being report based; the OA counters are configured in HW as a set while perf generally expects counter configurations to be orthogonal. Although counters can be associated with a group leader as they are opened, there's no clear precedent for being able to provide group-wide configuration attributes (for example we want to let userspace choose the OA unit report format used to capture all counters in a set, or specify a GPU context to filter metrics on). We avoided using perf's grouping feature and forwarded OA reports to userspace via perf's 'raw' sample field. This suited our userspace well considering how coupled the counters are when dealing with normalizing. It would be inconvenient to split counters up into separate events, only to require userspace to recombine them. For Mesa it's also convenient to be forwarded raw, periodic reports for combining with the side-band raw reports it captures using `MI_REPORT_PERF_COUNT` commands.

- As a side note on perf's grouping feature; there was also some concern that using `PERF_FORMAT_GROUP` as a way to pack together counter values would quite drastically inflate our sample sizes, which would likely lower the effective sampling resolutions we could use when the available memory bandwidth is limited.

With the OA unit's report formats, counters are packed together as 32 or 40bit values, with the largest report size being 256 bytes.

`PERF_FORMAT_GROUP` values are 64bit, but there doesn't appear to be a documented ordering to the values, implying `PERF_FORMAT_ID` must also be used to add a 64bit ID before each value; giving 16 bytes per counter.

Related to counter orthogonality; we can't time share the OA unit, while event scheduling is a central design idea within perf for allowing userspace to open + enable more events than can be configured in HW at any one time. The OA unit is not designed to allow re-configuration while in use. We can't reconfigure the OA unit without losing internal OA unit state which we can't access explicitly to save and restore. Reconfiguring the OA unit is also relatively slow, involving ~100 register writes. From userspace Mesa also depends on a stable OA configuration when emitting `MI_REPORT_PERF_COUNT` commands and importantly the OA unit can't be disabled while there are outstanding `MI_RPC` commands lest we hang the command streamer.

The contents of sample records aren't extensible by device drivers (i.e. the `sample_type` bits). As an example; Sourab Gupta had been looking to attach GPU timestamps to our OA samples. We were shoehorning OA reports into sample records by using the 'raw' field, but

it's tricky to pack more than one thing into this field because events/core.c currently only lets a pmu give a single raw data pointer plus len which will be copied into the ring buffer. To include more than the OA report we'd have to copy the report into an intermediate larger buffer. I'd been considering allowing a vector of data+len values to be specified for copying the raw data, but it felt like a kludge to being using the raw field for this purpose.

- It felt like our perf based PMU was making some technical compromises just for the sake of using perf:

perf_event_open() requires events to either relate to a pid or a specific cpu core, while our device pmu related to neither. Events opened with a pid will be automatically enabled/disabled according to the scheduling of that process - so not appropriate for us. When an event is related to a cpu id, perf ensures pmu methods will be invoked via an inter process interrupt on that core. To avoid invasive changes our userspace opened OA perf events for a specific cpu. This was workable but it meant the majority of the OA driver ran in atomic context, including all OA report forwarding, which wasn't really necessary in our case and seems to make our locking requirements somewhat complex as we handled the interaction with the rest of the i915 driver.

i915 Driver Entry Points

This section covers the entypoints exported outside of i915_perf.c to integrate with drm/i915 and to handle the *DRM_I915_PERF_OPEN* ioctl.

int **i915_perf_init**(struct drm_i915_private *i915)
 initialize i915-perf state on module bind

Parameters

struct drm_i915_private *i915
 i915 device instance

Description

Initializes i915-perf state without exposing anything to userspace.

Note

i915-perf initialization is split into an 'init' and 'register' phase with the *i915_perf_register()* exposing state to userspace.

void **i915_perf_fini**(struct drm_i915_private *i915)
 Counter part to *i915_perf_init()*

Parameters

struct drm_i915_private *i915
 i915 device instance

void **i915_perf_register**(struct drm_i915_private *i915)
 exposes i915-perf to userspace

Parameters

struct drm_i915_private *i915
 i915 device instance

Description

In particular OA metric sets are advertised under a `sysfs metrics/` directory allowing userspace to enumerate valid IDs that can be used to open an i915-perf stream.

```
void i915_perf_unregister(struct drm_i915_private *i915)
    hide i915-perf from userspace
```

Parameters

```
struct drm_i915_private *i915
    i915 device instance
```

Description

i915-perf state cleanup is split up into an ‘unregister’ and ‘deinit’ phase where the interface is first hidden from userspace by `i915_perf_unregister()` before cleaning up remaining state in `i915_perf_fini()`.

```
int i915_perf_open_ioctl(struct drm_device *dev, void *data, struct drm_file *file)
    DRM ioctl() for userspace to open a stream FD
```

Parameters

```
struct drm_device *dev
    drm device

void *data
    ioctl data copied from userspace (unvalidated)

struct drm_file *file
    drm file
```

Description

Validates the stream open parameters given by userspace including flags and an array of u64 key, value pair properties.

Very little is assumed up front about the nature of the stream being opened (for instance we don’t assume it’s for periodic OA unit metrics). An i915-perf stream is expected to be a suitable interface for other forms of buffered data written by the GPU besides periodic OA metrics.

Note we copy the properties from userspace outside of the i915 perf mutex to avoid an awkward lockdep with `mmap_lock`.

Most of the implementation details are handled by `i915_perf_open_ioctl_locked()` after taking the `gt->perf.lock` mutex for serializing with any non-file-operation driver hooks.

Return

A newly opened i915 Perf stream file descriptor or negative error code on failure.

```
int i915_perf_release(struct inode *inode, struct file *file)
    handles userspace close() of a stream file
```

Parameters

```
struct inode *inode
    anonymous inode associated with file

struct file *file
    An i915 perf stream file
```

Description

Cleans up any resources associated with an open i915 perf stream file.

NB: close() can't really fail from the userspace point of view.

Return

zero on success or a negative error code.

int **i915_perf_add_config_ioctl**(struct *drm_device* *dev, void *data, struct *drm_file* *file)
DRM ioctl() for userspace to add a new OA config

Parameters

struct drm_device *dev
drm device

void *data
ioctl data (pointer to *struct drm_i915_perf_oa_config*) copied from userspace (unvalidated)

struct drm_file *file
drm file

Description

Validates the submitted OA register to be saved into a new OA config that can then be used for programming the OA unit and its NOA network.

Return

A new allocated config number to be used with the perf open ioctl or a negative error code on failure.

int **i915_perf_remove_config_ioctl**(struct *drm_device* *dev, void *data, struct *drm_file* *file)
DRM ioctl() for userspace to remove an OA config

Parameters

struct drm_device *dev
drm device

void *data
ioctl data (pointer to u64 integer) copied from userspace

struct drm_file *file
drm file

Description

Configs can be removed while being used, they will stop appearing in sysfs and their content will be freed when the stream using the config is closed.

Return

0 on success or a negative error code on failure.

i915 Perf Stream

This section covers the stream-semantics-agnostic structures and functions for representing an i915 perf stream FD and associated file operations.

struct **i915_perf_stream**

state for a single open stream FD

Definition:

```
struct i915_perf_stream {
    struct i915_perf *perf;
    struct intel_uncore *uncore;
    struct intel_engine_cs *engine;
    struct mutex lock;
    u32 sample_flags;
    int sample_size;
    struct i915_gem_context *ctx;
    bool enabled;
    bool hold_preemption;
    const struct i915_perf_stream_ops *ops;
    struct i915_oa_config *oa_config;
    struct llist_head oa_config_bos;
    struct intel_context *pinned_ctx;
    u32 specific_ctx_id;
    u32 specific_ctx_id_mask;
    struct hrtimer poll_check_timer;
    wait_queue_head_t poll_wq;
    bool pollin;
    bool periodic;
    int period_exponent;
    struct {
        const struct i915_oa_format *format;
        struct i915_vma *vma;
        u8 *vaddr;
        u32 last_ctx_id;
        int size_exponent;
        spinlock_t ptr_lock;
        u32 head;
        u32 tail;
    } oa_buffer;
    struct i915_vma *noa_wait;
    u64 poll_oa_period;
    bool override_gucrc;
};
```

Members

perf

i915_perf backpointer

uncore

mmio access path

engine

Engine associated with this performance stream.

lock

Lock associated with operations on stream

sample_flags

Flags representing the *DRM_I915_PERF_PROP_SAMPLE_** properties given when opening a stream, representing the contents of a single sample as read() by userspace.

sample_size

Considering the configured contents of a sample combined with the required header size, this is the total size of a single sample record.

ctx

NULL if measuring system-wide across all contexts or a specific context that is being monitored.

enabled

Whether the stream is currently enabled, considering whether the stream was opened in a disabled state and based on *I915_PERF_IOCTL_ENABLE* and *I915_PERF_IOCTL_DISABLE* calls.

hold_preemption

Whether preemption is put on hold for command submissions done on the **ctx**. This is useful for some drivers that cannot easily post process the OA buffer context to subtract delta of performance counters not associated with **ctx**.

ops

The callbacks providing the implementation of this specific type of configured stream.

oa_config

The OA configuration used by the stream.

oa_config_bos

A list of struct *i915_oa_config_bo* allocated lazily each time **oa_config** changes.

pinned_ctx

The OA context specific information.

specific_ctx_id

The id of the specific context.

specific_ctx_id_mask

The mask used to masking *specific_ctx_id* bits.

poll_check_timer

High resolution timer that will periodically check for data in the circular OA buffer for notifying userspace (e.g. during a read() or poll()).

poll_wq

The wait queue that hrtimer callback wakes when it sees data ready to read in the circular OA buffer.

pollin

Whether there is data available to read.

periodic

Whether periodic sampling is currently enabled.

period_exponent

The OA unit sampling frequency is derived from this.

oa_buffer

State of the OA buffer.

noa_wait

A batch buffer doing a wait on the GPU for the NOA logic to be reprogrammed.

poll_oa_period

The period in nanoseconds at which the OA buffer should be checked for available data.

override_gucrc

GuC RC has been overridden for the perf stream, and we need to restore the default configuration on release.

struct i915_perf_stream_ops

the OPs to support a specific stream type

Definition:

```
struct i915_perf_stream_ops {
    void (*enable)(struct i915_perf_stream *stream);
    void (*disable)(struct i915_perf_stream *stream);
    void (*poll_wait)(struct i915_perf_stream *stream, struct file *file, poll_
→table *wait);
    int (*wait_unlocked)(struct i915_perf_stream *stream);
    int (*read)(struct i915_perf_stream *stream, char __user *buf, size_t count,
→size_t *offset);
    void (*destroy)(struct i915_perf_stream *stream);
};
```

Members

enable

Enables the collection of HW samples, either in response to *I915_PERF_IOCTL_ENABLE* or implicitly called when stream is opened without *I915_PERF_FLAG_DISABLED*.

disable

Disables the collection of HW samples, either in response to *I915_PERF_IOCTL_DISABLE* or implicitly called before destroying the stream.

poll_wait

Call `poll_wait`, passing a wait queue that will be woken once there is something ready to `read()` for the stream

wait_unlocked

For handling a blocking read, wait until there is something to ready to `read()` for the stream. E.g. wait on the same wait queue that would be passed to `poll_wait()`.

read

Copy buffered metrics as records to userspace **buf**: the userspace, destination buffer **count**: the number of bytes to copy, requested by userspace **offset**: zero at the start of the read, updated as the read proceeds, it represents how many bytes have been copied so far and the buffer offset for copying the next record.

Copy as many buffered i915 perf samples and records for this stream to userspace as will fit in the given buffer.

Only write complete records; returning `-ENOSPC` if there isn't room for a complete record.

Return any error condition that results in a short read such as `-ENOSPC` or `-EFAULT`, even though these may be squashed before returning to userspace.

destroy

Cleanup any stream specific resources.

The stream will always be disabled before this is called.

int **read_properties_unlocked**(struct i915_perf *perf, u64 __user *uprops, u32 n_props,
 struct *perf_open_properties* *props)

validate + copy userspace stream open properties

Parameters

struct i915_perf *perf
i915 perf instance

u64 __user *uprops
The array of u64 key value pairs given by userspace

u32 n_props
The number of key value pairs expected in **uprops**

struct perf_open_properties *props
The stream configuration built up while validating properties

Description

Note this function only validates properties in isolation it doesn't validate that the combination of properties makes sense or that all properties necessary for a particular kind of stream have been set.

Note that there currently aren't any ordering requirements for properties so we shouldn't validate or assume anything about ordering here. This doesn't rule out defining new properties with ordering requirements in the future.

int **i915_perf_open_ioctl_locked**(struct i915_perf *perf, struct drm_i915_perf_open_param
 *param, struct *perf_open_properties* *props, struct
 drm_file *file)

DRM ioctl() for userspace to open a stream FD

Parameters

struct i915_perf *perf
i915 perf instance

struct drm_i915_perf_open_param *param
The open parameters passed to 'DRM_I915_PERF_OPEN'

struct perf_open_properties *props
individually validated u64 property value pairs

struct drm_file *file
drm file

Description

See `i915_perf_ioctl_open()` for interface details.

Implements further stream config validation and stream initialization on behalf of `i915_perf_open_ioctl()` with the `gt->perf.lock` mutex taken to serialize with any non-file-operation driver hooks.

In the case where userspace is interested in OA unit metrics then further config validation and stream initialization details will be handled by `i915_oa_stream_init()`. The code here should only validate config state that will be relevant to all stream types / backends.

Note

at this point the **props** have only been validated in isolation and it's still necessary to validate that the combination of properties makes sense.

Return

zero on success or a negative error code.

`void i915_perf_destroy_locked(struct i915_perf_stream *stream)`
destroy an i915 perf stream

Parameters

`struct i915_perf_stream *stream`
An i915 perf stream

Description

Frees all resources associated with the given i915 perf **stream**, disabling any associated data capture in the process.

Note

The `gt->perf.lock` mutex has been taken to serialize with any non-file-operation driver hooks.

`ssize_t i915_perf_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)`
handles read() FOP for i915 perf stream FDs

Parameters

`struct file *file`
An i915 perf stream file

`char __user *buf`
destination buffer given by userspace

`size_t count`
the number of bytes userspace wants to read

`loff_t *ppos`
(inout) file seek position (unused)

Description

The entry point for handling a read() on a stream file descriptor from userspace. Most of the work is left to the `i915_perf_read_locked()` and `i915_perf_stream_ops->read` but to save having stream implementations (of which we might have multiple later) we handle blocking read here.

We can also consistently treat trying to read from a disabled stream as an IO error so implementations can assume the stream is enabled while reading.

Return

The number of bytes copied or a negative error code on failure.

long **i915_perf_ioctl**(struct *file* *file, unsigned int cmd, unsigned long arg)
support ioctl() usage with i915 perf stream FDs

Parameters

struct file *file
An i915 perf stream file

unsigned int cmd
the ioctl request

unsigned long arg
the ioctl data

Description

Implementation deferred to *i915_perf_ioctl_locked()*.

Return

zero on success or a negative error code. Returns -EINVAL for an unknown ioctl request.

void **i915_perf_enable_locked**(struct *i915_perf_stream* *stream)
handle *I915_PERF_IOCTL_ENABLE* ioctl

Parameters

struct i915_perf_stream *stream
A disabled i915 perf stream

Description

[Re]enables the associated capture of data for this stream.

If a stream was previously enabled then there's currently no intention to provide userspace any guarantee about the preservation of previously buffered data.

void **i915_perf_disable_locked**(struct *i915_perf_stream* *stream)
handle *I915_PERF_IOCTL_DISABLE* ioctl

Parameters

struct i915_perf_stream *stream
An enabled i915 perf stream

Description

Disables the associated capture of data for this stream.

The intention is that disabling an re-enabling a stream will ideally be cheaper than destroying and re-opening a stream with the same configuration, though there are no formal guarantees about what state or buffered data must be retained between disabling and re-enabling a stream.

Note

while a stream is disabled it's considered an error for userspace to attempt to read from the stream (-EIO).

`__poll_t i915_perf_poll(struct file *file, poll_table *wait)`
call `poll_wait()` with a suitable wait queue for stream

Parameters

struct file *file
An i915 perf stream file

poll_table *wait
poll() state table

Description

For handling userspace polling on an i915 perf stream, this ensures `poll_wait()` gets called with a wait queue that will be woken for new stream data.

Note

Implementation deferred to `i915_perf_poll_locked()`

Return

any poll events that are ready without sleeping

`__poll_t i915_perf_poll_locked(struct i915_perf_stream *stream, struct file *file, poll_table *wait)`
poll_wait() with a suitable wait queue for stream

Parameters

struct i915_perf_stream *stream
An i915 perf stream

struct file *file
An i915 perf stream file

poll_table *wait
poll() state table

Description

For handling userspace polling on an i915 perf stream, this calls through to `i915_perf_stream_ops->poll_wait` to call `poll_wait()` with a wait queue that will be woken for new stream data.

Return

any poll events that are ready without sleeping

i915 Perf Observation Architecture Stream

struct **i915_oa_ops**

Gen specific implementation of an OA unit stream

Definition:

```
struct i915_oa_ops {
    bool (*is_valid_b_counter_reg)(struct i915_perf *perf, u32 addr);
    bool (*is_valid_mux_reg)(struct i915_perf *perf, u32 addr);
    bool (*is_valid_flex_reg)(struct i915_perf *perf, u32 addr);
    int (*enable_metric_set)(struct i915_perf_stream *stream, struct i915_
    ↪active *active);
    void (*disable_metric_set)(struct i915_perf_stream *stream);
    void (*oa_enable)(struct i915_perf_stream *stream);
    void (*oa_disable)(struct i915_perf_stream *stream);
    int (*read)(struct i915_perf_stream *stream, char __user *buf, size_t count,
    ↪size_t *offset);
    u32 (*oa_hw_tail_read)(struct i915_perf_stream *stream);
};
```

Members

is_valid_b_counter_reg

Validates register's address for programming boolean counters for a particular platform.

is_valid_mux_reg

Validates register's address for programming mux for a particular platform.

is_valid_flex_reg

Validates register's address for programming flex EU filtering for a particular platform.

enable_metric_set

Selects and applies any MUX configuration to set up the Boolean and Custom (B/C) counters that are part of the counter reports being sampled. May apply system constraints such as disabling EU clock gating as required.

disable_metric_set

Remove system constraints associated with using the OA unit.

oa_enable

Enable periodic sampling

oa_disable

Disable periodic sampling

read

Copy data from the circular OA buffer into a given userspace buffer.

oa_hw_tail_read

read the OA tail pointer register

In particular this enables us to share all the fiddly code for handling the OA unit tail pointer race that affects multiple generations.

int **i915_oa_stream_init**(struct *i915_perf_stream* *stream, struct
drm_i915_perf_open_param *param, struct *perf_open_properties*
*props)

validate combined props for OA stream and init

Parameters

struct i915_perf_stream *stream

An i915 perf stream

struct drm_i915_perf_open_param *param

The open parameters passed to *DRM_I915_PERF_OPEN*

struct perf_open_properties *props

The property state that configures stream (individually validated)

Description

While *read_properties_unlocked()* validates properties in isolation it doesn't ensure that the combination necessarily makes sense.

At this point it has been determined that userspace wants a stream of OA metrics, but still we need to further validate the combined properties are OK.

If the configuration makes sense then we can allocate memory for a circular OA buffer and apply the requested metric set configuration.

Return

zero on success or a negative error code.

int **i915_oa_read**(struct *i915_perf_stream* *stream, char __user *buf, size_t count, size_t *offset)

just calls through to *i915_oa_ops->read*

Parameters

struct i915_perf_stream *stream

An i915-perf stream opened for OA metrics

char __user *buf

destination buffer given by userspace

size_t count

the number of bytes userspace wants to read

size_t *offset

(inout): the current position for writing into **buf**

Description

Updates **offset** according to the number of bytes successfully copied into the userspace buffer.

Return

zero on success or a negative error code

void **i915_oa_stream_enable**(struct *i915_perf_stream* *stream)

handle *I915_PERF_IOCTL_ENABLE* for OA stream

Parameters

struct i915_perf_stream *stream

An i915 perf stream opened for OA metrics

Description

[Re]enables hardware periodic sampling according to the period configured when opening the stream. This also starts a hrtimer that will periodically check for data in the circular OA buffer for notifying userspace (e.g. during a read() or poll()).

```
void i915_oa_stream_disable(struct i915_perf_stream *stream)
    handle I915_PERF_IOCTL_DISABLE for OA stream
```

Parameters

```
struct i915_perf_stream *stream
    An i915 perf stream opened for OA metrics
```

Description

Stops the OA unit from periodically writing counter reports into the circular OA buffer. This also stops the hrtimer that periodically checks for data in the circular OA buffer, for notifying userspace.

```
int i915_oa_wait_unlocked(struct i915_perf_stream *stream)
    handles blocking IO until OA data available
```

Parameters

```
struct i915_perf_stream *stream
    An i915-perf stream opened for OA metrics
```

Description

Called when userspace tries to read() from a blocking stream FD opened for OA metrics. It waits until the hrtimer callback finds a non-empty OA buffer and wakes us.

Note

it's acceptable to have this return with some false positives since any subsequent read handling will return -EAGAIN if there isn't really data ready for userspace yet.

Return

zero on success or a negative error code

```
void i915_oa_poll_wait(struct i915_perf_stream *stream, struct file *file, poll_table *wait)
    call poll_wait() for an OA stream poll()
```

Parameters

```
struct i915_perf_stream *stream
    An i915-perf stream opened for OA metrics
```

```
struct file *file
    An i915 perf stream file
```

```
poll_table *wait
    poll() state table
```

Description

For handling userspace polling on an i915 perf stream opened for OA metrics, this starts a poll_wait with the wait queue that our hrtimer callback wakes when it sees data ready to read in the circular OA buffer.

Other i915 Perf Internals

This section simply includes all other currently documented i915 perf internals, in no particular order, but may include some more minor utilities or platform specific details than found in the more high-level sections.

struct **perf_open_properties**

for validated properties given to open a stream

Definition:

```
struct perf_open_properties {
    u32 sample_flags;
    u64 single_context:1;
    u64 hold_preemption:1;
    u64 ctx_handle;
    int metrics_set;
    int oa_format;
    bool oa_periodic;
    int oa_period_exponent;
    struct intel_engine_cs *engine;
    bool has_sseu;
    struct intel_sseu sseu;
    u64 poll_oa_period;
};
```

Members

sample_flags

*DRM_I915_PERF_PROP_SAMPLE_** properties are tracked as flags

single_context

Whether a single or all gpu contexts should be monitored

hold_preemption

Whether the preemption is disabled for the filtered context

ctx_handle

A gem ctx handle for use with **single_context**

metrics_set

An ID for an OA unit metric set advertised via sysfs

oa_format

An OA unit HW report format

oa_periodic

Whether to enable periodic OA unit sampling

oa_period_exponent

The OA unit sampling period is derived from this

engine

The engine (typically rcs0) being monitored by the OA unit

has_sseu

Whether **sseu** was specified by userspace

sseu

internal SSEU configuration computed either from the userspace specified configuration in the opening parameters or a default value (see `get_default_sseu_config()`)

poll_oa_period

The period in nanoseconds at which the CPU will check for OA data availability

Description

As `read_properties_unlocked()` enumerates and validates the properties given to open a stream of metrics the configuration is built up in the structure which starts out zero initialized.

bool **oa_buffer_check_unlocked**(struct *i915_perf_stream* *stream)

check for data and update tail ptr state

Parameters

struct *i915_perf_stream* *stream

i915 stream instance

Description

This is either called via fops (for blocking reads in user ctx) or the poll check hrtimer (atomic ctx) to check the OA buffer tail pointer and check if there is data available for userspace to read.

This function is central to providing a workaround for the OA unit tail pointer having a race with respect to what data is visible to the CPU. It is responsible for reading tail pointers from the hardware and giving the pointers time to ‘age’ before they are made available for reading. (See description of `OA_TAIL_MARGIN_NSEC` above for further details.)

Besides returning true when there is data available to read() this function also updates the tail in the `oa_buffer` object.

Note

It’s safe to read OA config state here unlocked, assuming that this is only called while the stream is enabled, while the global OA configuration can’t be modified.

Return

true if the OA buffer contains data, else false

int **append_oa_status**(struct *i915_perf_stream* *stream, char __user *buf, size_t count, size_t *offset, enum *drm_i915_perf_record_type* type)

Appends a status record to a userspace read() buffer.

Parameters

struct *i915_perf_stream* *stream

An i915-perf stream opened for OA metrics

char __user *buf

destination buffer given by userspace

size_t count

the number of bytes userspace wants to read

size_t *offset

(inout): the current position for writing into **buf**

enum `drm_i915_perf_record_type` type

The kind of status to report to userspace

Description

Writes a status record (such as `DRM_I915_PERF_RECORD_OA_REPORT_LOST`) into the userspace `read()` buffer.

The **buf offset** will only be updated on success.

Return

0 on success, negative error code on failure.

int **append_oa_sample**(struct *i915_perf_stream* *stream, char __user *buf, size_t count, size_t *offset, const u8 *report)

Copies single OA report into userspace `read()` buffer.

Parameters

struct *i915_perf_stream* *stream

An i915-perf stream opened for OA metrics

char __user *buf

destination buffer given by userspace

size_t count

the number of bytes userspace wants to read

size_t *offset

(inout): the current position for writing into **buf**

const u8 *report

A single OA report to (optionally) include as part of the sample

Description

The contents of a sample are configured through `DRM_I915_PERF_PROP_SAMPLE_*` properties when opening a stream, tracked as `stream->sample_flags`. This function copies the requested components of a single sample to the given `read()` **buf**.

The **buf offset** will only be updated on success.

Return

0 on success, negative error code on failure.

int **gen8_append_oa_reports**(struct *i915_perf_stream* *stream, char __user *buf, size_t count, size_t *offset)

Copies all buffered OA reports into userspace `read()` buffer.

Parameters

struct *i915_perf_stream* *stream

An i915-perf stream opened for OA metrics

char __user *buf

destination buffer given by userspace

size_t count

the number of bytes userspace wants to read

size_t *offset

(inout): the current position for writing into **buf**

Description

Notably any error condition resulting in a short read (-ENOSPC or -EFAULT) will be returned even though one or more records may have been successfully copied. In this case it's up to the caller to decide if the error should be squashed before returning to userspace.

Note

reports are consumed from the head, and appended to the tail, so the tail chases the head?... If you think that's mad and back-to-front you're not alone, but this follows the Gen PRM naming convention.

Return

0 on success, negative error code on failure.

int **gen8_oa_read**(struct *i915_perf_stream* *stream, char __user *buf, size_t count, size_t *offset)

copy status records then buffered OA reports

Parameters

struct i915_perf_stream *stream

An i915-perf stream opened for OA metrics

char __user *buf

destination buffer given by userspace

size_t count

the number of bytes userspace wants to read

size_t *offset

(inout): the current position for writing into **buf**

Description

Checks OA unit status registers and if necessary appends corresponding status records for userspace (such as for a buffer full condition) and then initiate appending any buffered OA reports.

Updates **offset** according to the number of bytes successfully copied into the userspace buffer.

NB: some data may be successfully copied to the userspace buffer even if an error is returned, and this is reflected in the updated **offset**.

Return

zero on success or a negative error code

int **gen7_append_oa_reports**(struct *i915_perf_stream* *stream, char __user *buf, size_t count, size_t *offset)

Copies all buffered OA reports into userspace read() buffer.

Parameters

struct i915_perf_stream *stream

An i915-perf stream opened for OA metrics

char __user *buf

destination buffer given by userspace

size_t count

the number of bytes userspace wants to read

size_t *offset

(inout): the current position for writing into **buf**

Description

Notably any error condition resulting in a short read (-ENOSPC or -EFAULT) will be returned even though one or more records may have been successfully copied. In this case it's up to the caller to decide if the error should be squashed before returning to userspace.

Note

reports are consumed from the head, and appended to the tail, so the tail chases the head?... If you think that's mad and back-to-front you're not alone, but this follows the Gen PRM naming convention.

Return

0 on success, negative error code on failure.

int **gen7_oa_read**(struct *i915_perf_stream* *stream, char __user *buf, size_t count, size_t *offset)

copy status records then buffered OA reports

Parameters

struct i915_perf_stream *stream

An i915-perf stream opened for OA metrics

char __user *buf

destination buffer given by userspace

size_t count

the number of bytes userspace wants to read

size_t *offset

(inout): the current position for writing into **buf**

Description

Checks Gen 7 specific OA unit status registers and if necessary appends corresponding status records for userspace (such as for a buffer full condition) and then initiate appending any buffered OA reports.

Updates **offset** according to the number of bytes successfully copied into the userspace buffer.

Return

zero on success or a negative error code

int **oa_get_render_ctx_id**(struct *i915_perf_stream* *stream)

determine and hold ctx hw id

Parameters

struct i915_perf_stream *stream

An i915-perf stream opened for OA metrics

Description

Determine the render context hw id, and ensure it remains fixed for the lifetime of the stream. This ensures that we don't have to worry about updating the context ID in OACONTROL on the fly.

Return

zero on success or a negative error code

void **oa_put_render_ctx_id**(struct *i915_perf_stream* *stream)
counterpart to oa_get_render_ctx_id releases hold

Parameters

struct i915_perf_stream *stream
An i915-perf stream opened for OA metrics

Description

In case anything needed doing to ensure the context HW ID would remain valid for the lifetime of the stream, then that can be undone here.

long **i915_perf_ioctl_locked**(struct *i915_perf_stream* *stream, unsigned int cmd, unsigned long arg)
support ioctl() usage with i915 perf stream FDs

Parameters

struct i915_perf_stream *stream
An i915 perf stream

unsigned int cmd
the ioctl request

unsigned long arg
the ioctl data

Return

zero on success or a negative error code. Returns -EINVAL for an unknown ioctl request.

int **i915_perf_ioctl_version**(struct drm_i915_private *i915)
Version of the i915-perf subsystem

Parameters

struct drm_i915_private *i915
The i915 device

Description

This version number is used by userspace to detect available features.

10.2.8 Style

The drm/i915 driver codebase has some style rules in addition to (and, in some cases, deviating from) the kernel coding style.

Register macro definition style

The style guide for `i915_reg.h`.

Follow the style described here for new macros, and while changing existing macros. Do **not** mass change existing definitions just to update the style.

File Layout

Keep helper macros near the top. For example, `_PIPE()` and friends.

Prefix macros that generally should not be used outside of this file with underscore `'_'`. For example, `_PIPE()` and friends, single instances of registers that are defined solely for the use by function-like macros.

Avoid using the underscore prefixed macros outside of this file. There are exceptions, but keep them to a minimum.

There are two basic types of register definitions: Single registers and register groups. Register groups are registers which have two or more instances, for example one per pipe, port, transcoder, etc. Register groups should be defined using function-like macros.

For single registers, define the register offset first, followed by register contents.

For register groups, define the register instance offsets first, prefixed with underscore, followed by a function-like macro choosing the right instance based on the parameter, followed by register contents.

Define the register contents (i.e. bit and bit field macros) from most significant to least significant bit. Indent the register content macros using two extra spaces between `#define` and the macro name.

Define bit fields using `REG_GENMASK(h, l)`. Define bit field contents using `REG_FIELD_PREP(mask, value)`. This will define the values already shifted in place, so they can be directly OR'd together. For convenience, function-like macros may be used to define bit fields, but do note that the macros may be needed to read as well as write the register contents.

Define bits using `REG_BIT(N)`. Do **not** add `_BIT` suffix to the name.

Group the register and its contents together without blank lines, separate from other registers and their contents with one blank line.

Indent macro values from macro names using TABs. Align values vertically. Use braces in macro values as needed to avoid unintended precedence after macro substitution. Use spaces in macro values according to kernel coding style. Use lower case in hexadecimal values.

Naming

Try to name registers according to the specs. If the register name changes in the specs from platform to another, stick to the original name.

Try to re-use existing register macro definitions. Only add new macros for new register offsets, or when the register contents have changed enough to warrant a full redefinition.

When a register macro changes for a new platform, prefix the new macro using the platform acronym or generation. For example, SKL_ or GEN8_. The prefix signifies the start platform/generation using the register.

When a bit (field) macro changes or gets added for a new platform, while retaining the existing register macro, add a platform acronym or generation suffix to the name. For example, _SKL or _GEN8.

Examples

(Note that the values in the example are indented using spaces instead of TABs to avoid misalignment in generated documentation. Use TABs in the definitions.):

```
#define _F00_A          0xf000
#define _F00_B          0xf001
#define F00(pipe)       _MMIO_PIPE(pipe, _F00_A, _F00_B)
#define F00_ENABLE      REG_BIT(31)
#define F00_MODE_MASK   REG_GENMASK(19, 16)
#define F00_MODE_BAR    REG_FIELD_PREP(F00_MODE_MASK, 0)
#define F00_MODE_BAZ    REG_FIELD_PREP(F00_MODE_MASK, 1)
#define F00_MODE_QUIX_SNB REG_FIELD_PREP(F00_MODE_MASK, 2)

#define BAR              _MMIO(0xb000)
#define GEN8_BAR         _MMIO(0xb888)
```

10.2.9 i915 DRM client usage stats implementation

The drm/i915 driver implements the DRM client usage stats specification as documented in [DRM client usage stats](#).

Example of the output showing the implemented key value pairs and entirety of the currently possible format options:

```
pos:      0
flags:    0100002
mnt_id:   21
drm-driver: i915
drm-pdev: 0000:00:02.0
drm-client-id: 7
drm-engine-render: 9288864723 ns
drm-engine-copy: 2035071108 ns
drm-engine-video: 0 ns
```

```
drm-engine-capacity-video:    2
drm-engine-video-enhance:    0 ns
```

Possible *drm-engine*- key names are: *render*, *copy*, *video* and *video-enhance*.

10.3 drm/mcde ST-Ericsson MCDE Multi-channel display engine

The MCDE (short for multi-channel display engine) is a graphics controller found in the Ux500 chipsets, such as NovaThor U8500. It was initially conceptualized by ST Microelectronics for the successor of the Nomadik line, STn8500 but productified in the ST-Ericsson U8500 where it was used for mass-market deployments in Android phones from Samsung and Sony Ericsson.

It can do 1080p30 on SDTV CCIR656, DPI-2, DBI-2 or DSI for panels with or without frame buffering and can convert most input formats including most variants of RGB and YUV.

The hardware has four display pipes, and the layout is a little bit like this:

Memory	-> Overlay	-> Channel	-> FIFO	-> 8 formatters	-> DSI/DPI
External	0..5	0..3	A,B,	6 x DSI	bridge
source 0..9			C0,C1	2 x DPI	

FIFOs A and B are for LCD and HDMI while FIFO C0/C1 are for panels with embedded buffer. 6 of the formatters are for DSI, 3 pairs for VID/CMD respectively. 2 of the formatters are for DPI.

Behind the formatters are the DSI or DPI ports that route to the external pins of the chip. As there are 3 DSI ports and one DPI port, it is possible to configure up to 4 display pipelines (effectively using channels 0..3) for concurrent use.

In the current DRM/KMS setup, we use one external source, one overlay, one FIFO and one formatter which we connect to the simple DMA framebuffer helpers. We then provide a bridge to the DSI port, and on the DSI port bridge we connect hang a panel bridge or other bridge. This may be subject to change as we exploit more of the hardware capabilities.

TODO:

- Enabled damaged rectangles using `drm_plane_enable_fb_damage_clips()` so we can selectively just transmit the damaged area to a command-only display.
- Enable mixing of more planes, possibly at the cost of moving away from using the simple framebuffer pipeline.
- Enable output to bridges such as the AV8100 HDMI encoder from the DSI bridge.

10.4 drm/meson AmLogic Meson Video Processing Unit

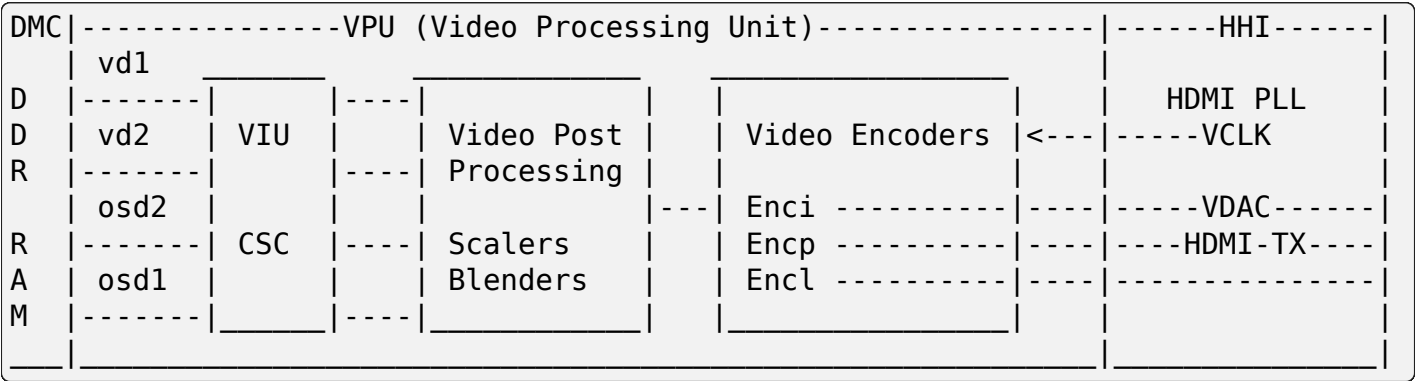
VPU Handles the Global Video Processing, it includes management of the clocks gates, blocks reset lines and power domains.

What is missing :

- Full reset of entire video processing HW blocks
- Scaling and setup of the VPU clock
- Bus clock gates
- Powering up video processing HW blocks
- Powering Up HDMI controller and PHY

10.4.1 Video Processing Unit

The Amlogic Meson Display controller is composed of several components that are going to be documented below:



10.4.2 Video Input Unit

VIU Handles the Pixel scanout and the basic Colorspace conversions We handle the following features :

- OSD1 RGB565/RGB888/xRGB8888 scanout
- RGB conversion to x/cb/cr
- Progressive or Interlace buffer scanout
- OSD1 Commit on Vsync
- HDR OSD matrix for GXL/GXM

What is missing :

- BGR888/xBGR8888/BGRx8888/BGRx8888 modes
- YUV4:2:2 Y0CbY1Cr scanout
- Conversion to YUV 4:4:4 from 4:2:2 input
- Colorkey Alpha matching

- Big endian scanout
- X/Y reverse scanout
- Global alpha setup
- OSD2 support, would need interlace switching on vsync
- OSD1 full scaling to support TV overscan

10.4.3 Video Post Processing

VPP Handles all the Post Processing after the Scanout from the VIU We handle the following post processings :

- **Postblend, Blends the OSD1 only**
We exclude OSD2, VS1, VS1 and Preblend output
- **Vertical OSD Scaler for OSD1 only, we disable vertical scaler and use it only for interlace scanout**
- Intermediate FIFO with default Amlogic values

What is missing :

- Preblend for video overlay pre-scaling
- OSD2 support for cursor framebuffer
- Video pre-scaling before postblend
- Full Vertical/Horizontal OSD scaling to support TV overscan
- HDR conversion

10.4.4 Video Encoder

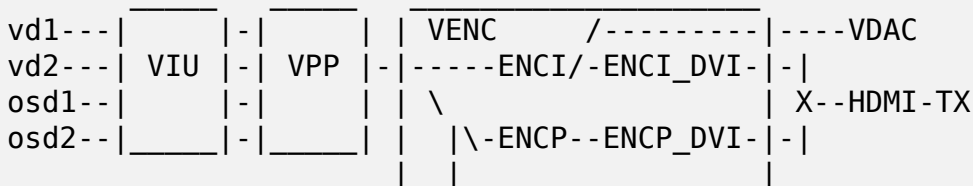
VENC Handle the pixels encoding to the output formats. We handle the following encodings :

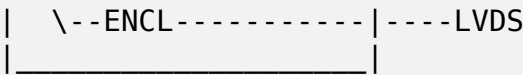
- CVBS Encoding via the ENCI encoder and VDAC digital to analog converter
- TMDS/HDMI Encoding via ENCI_DIV and ENCP
- Setup of more clock rates for HDMI modes

What is missing :

- LCD Panel encoding via ENCL
- TV Panel encoding via ENCT

VENC paths :





The ENCI is designed for PAL or NTSC encoding and can go through the VDAC directly for CVBS encoding or through the ENCI_DVI encoder for HDMI. The ENCP is designed for Progressive encoding but can also generate 1080i interlaced pixels, and was initially designed to encode pixels for VDAC to output RGB ou YUV analog outputs. It's output is only used through the ENCP_DVI encoder for HDMI. The ENCL LVDS encoder is not implemented.

The ENCI and ENCP encoders needs specially defined parameters for each supported mode and thus cannot be determined from standard video timings.

The ENCI end ENCP DVI encoders are more generic and can generate any timings from the pixel data generated by ENCI or ENCP, so can use the standard video timings are source for HW parameters.

10.4.5 Video Clocks

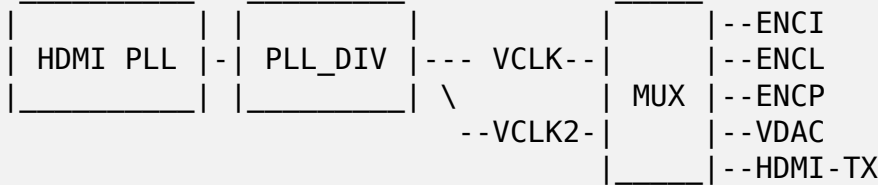
VCLK is the “Pixel Clock” frequency generator from a dedicated PLL. We handle the following encodings :

- CVBS 27MHz generator via the VCLK2 to the VENCI and VDAC blocks
- HDMI Pixel Clocks generation

What is missing :

- Genenate Pixel clocks for 2K/4K 10bit formats

Clock generator scheme :



Final clocks can take input for either VCLK or VCLK2, but VCLK is the preferred path for HDMI clocking and VCLK2 is the preferred path for CVBS VDAC clocking.

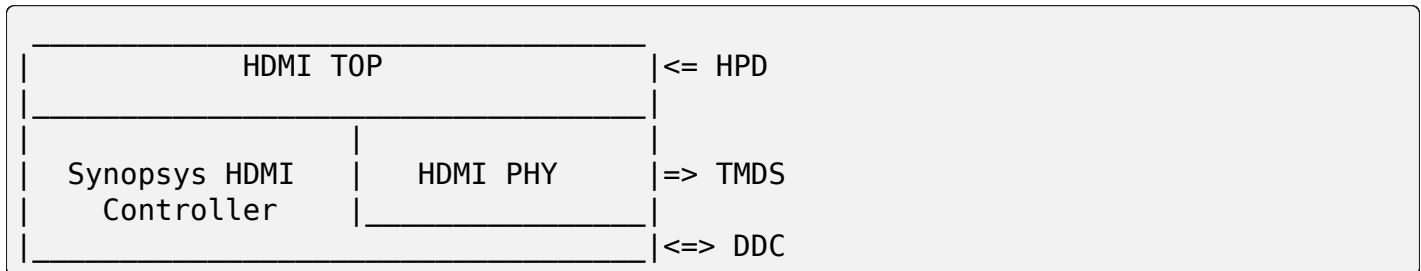
VCLK and VCLK2 have fixed divided clocks paths for /1, /2, /4, /6 or /12.

The PLL_DIV can achieve an additional fractional dividing like 1.5, 3.5, 3.75... to generate special 2K and 4K 10bit clocks.

10.4.6 HDMI Video Output

HDMI Output is composed of :

- A Synopsys DesignWare HDMI Controller IP
- A TOP control block controlling the Clocks and PHY
- A custom HDMI PHY in order convert video to TMDS signal



The HDMI TOP block only supports HPD sensing. The Synopsys HDMI Controller interrupt is routed through the TOP Block interrupt. Communication to the TOP Block and the Synopsys HDMI Controller is done a pair of addr+read/write registers. The HDMI PHY is configured by registers in the HHI register block.

Pixel data arrives in 4:4:4 format from the VENC block and the VPU HDMI mux selects either the ENCI encoder for the 576i or 480i formats or the ENCP encoder for all the other formats including interlaced HD formats. The VENC uses a DVI encoder on top of the ENCI or ENCP encoders to generate DVI timings for the HDMI controller.

GXBB, GXL and GXM embeds the Synopsys DesignWare HDMI TX IP version 2.01a with HDCP and I2C & S/PDIF audio source interfaces.

We handle the following features :

- HPD Rise & Fall interrupt
- HDMI Controller Interrupt
- HDMI PHY Init for 480i to 1080p60
- VENC & HDMI Clock setup for 480i to 1080p60
- VENC Mode setup for 480i to 1080p60

What is missing :

- PHY, Clock and Mode setup for 2k && 4k modes
- SDDC Scrambling mode for HDMI 2.0a
- HDCP Setup
- CEC Management

10.5 drm/pl111 ARM PrimeCell PL110 and PL111 CLCD Driver

The PL110/PL111 is a simple LCD controller that can support TFT and STN displays. This driver exposes a standard KMS interface for them.

The driver currently doesn't expose the cursor. The DRM API for cursors requires support for 64x64 ARGB8888 cursor images, while the hardware can only support 64x64 monochrome with masking cursors. While one could imagine trying to hack something together to look at the ARGB8888 and program reasonable in monochrome, we just don't expose the cursor at all instead, and leave cursor support to the application software cursor layer.

TODO:

- Fix race between setting plane base address and getting IRQ for vsync firing the pageflip completion.
- Read back hardware state at boot to skip reprogramming the hardware when doing a no-op modeset.
- Use the CLKSEL bit to support switching between the two external clock parents.

10.6 drm/tegra NVIDIA Tegra GPU and display driver

NVIDIA Tegra SoCs support a set of display, graphics and video functions via the host1x controller. host1x supplies command streams, gathered from a push buffer provided directly by the CPU, to its clients via channels. Software, or blocks amongst themselves, can use syncpoints for synchronization.

Up until, but not including, Tegra124 (aka Tegra K1) the drm/tegra driver supports the built-in GPU, comprised of the gr2d and gr3d engines. Starting with Tegra124 the GPU is based on the NVIDIA desktop GPU architecture and supported by the drm/nouveau driver.

The drm/tegra driver supports NVIDIA Tegra SoC generations since Tegra20. It has three parts:

- A host1x driver that provides infrastructure and access to the host1x services.
- A KMS driver that supports the display controllers as well as a number of outputs, such as RGB, HDMI, DSI, and DisplayPort.
- A set of custom userspace IOCTLs that can be used to submit jobs to the GPU and video engines via host1x.

10.6.1 Driver Infrastructure

The various host1x clients need to be bound together into a logical device in order to expose their functionality to users. The infrastructure that supports this is implemented in the host1x driver. When a driver is registered with the infrastructure it provides a list of compatible strings specifying the devices that it needs. The infrastructure creates a logical device and scan the device tree for matching device nodes, adding the required clients to a list. Drivers for individual clients register with the infrastructure as well and are added to the logical host1x device.

Once all clients are available, the infrastructure will initialize the logical device using a driver-provided function which will set up the bits specific to the subsystem and in turn initialize each of its clients.

Similarly, when one of the clients is unregistered, the infrastructure will destroy the logical device by calling back into the driver, which ensures that the subsystem specific bits are torn down and the clients destroyed in turn.

Host1x Infrastructure Reference

struct **host1x_bo_cache**

host1x buffer object cache

Definition:

```
struct host1x_bo_cache {
    struct list_head mappings;
    struct mutex lock;
};
```

Members

mappings

list of mappings

lock

synchronizes accesses to the list of mappings

Description

Note that entries are not periodically evicted from this cache and instead need to be explicitly released. This is used primarily for DRM/KMS where the cache's reference is released when the last reference to a buffer object represented by a mapping in this cache is dropped.

struct **host1x_client_ops**

host1x client operations

Definition:

```
struct host1x_client_ops {
    int (*early_init)(struct host1x_client *client);
    int (*init)(struct host1x_client *client);
    int (*exit)(struct host1x_client *client);
    int (*late_exit)(struct host1x_client *client);
    int (*suspend)(struct host1x_client *client);
    int (*resume)(struct host1x_client *client);
};
```

Members

early_init

host1x client early initialization code

init

host1x client initialization code

exit

host1x client tear down code

late_exit

host1x client late tear down code

suspend

host1x client suspend code

resume

host1x client resume code

struct host1x_client

host1x client structure

Definition:

```
struct host1x_client {
    struct list_head list;
    struct device *host;
    struct device *dev;
    struct iommu_group *group;
    const struct host1x_client_ops *ops;
    enum host1x_class class;
    struct host1x_channel *channel;
    struct host1x_syncpt **syncpts;
    unsigned int num_syncpts;
    struct host1x_client *parent;
    unsigned int usecount;
    struct mutex lock;
    struct host1x_bo_cache cache;
};
```

Members**list**

list node for the host1x client

host

pointer to struct device representing the host1x controller

dev

pointer to struct device backing this host1x client

group

IOMMU group that this client is a member of

ops

host1x client operations

class

host1x class represented by this client

channel

host1x channel associated with this client

syncpts

array of syncpoints requested for this client

num_syncpts

number of syncpoints requested for this client

parent

pointer to parent structure

usecount

reference count for this structure

lock

mutex for mutually exclusive concurrency

cache

host1x buffer object cache

struct host1x_driver

host1x logical device driver

Definition:

```
struct host1x_driver {
    struct device_driver driver;
    const struct of_device_id *subdevs;
    struct list_head list;
    int (*probe)(struct host1x_device *device);
    int (*remove)(struct host1x_device *device);
    void (*shutdown)(struct host1x_device *device);
};
```

Members**driver**

core driver

subdevs

table of OF device IDs matching subdevices for this driver

list

list node for the driver

probe

called when the host1x logical device is probed

remove

called when the host1x logical device is removed

shutdown

called when the host1x logical device is shut down

int **host1x_device_init**(struct host1x_device *device)

initialize a host1x logical device

Parameters

struct host1x_device *device

host1x logical device

Description

The driver for the host1x logical device can call this during execution of its [host1x_driver.probe](#) implementation to initialize each of its clients. The client drivers access the subsystem

specific driver data using the `host1x_client.parent` field and driver data associated with it (usually by calling `dev_get_drvdata()`).

int **host1x_device_exit**(struct host1x_device *device)
 uninitialize host1x logical device

Parameters

struct host1x_device *device
 host1x logical device

Description

When the driver for a host1x logical device is unloaded, it can call this function to tear down each of its clients. Typically this is done after a subsystem-specific data structure is removed and the functionality can no longer be used.

int **host1x_driver_register_full**(struct *host1x_driver* *driver, struct module *owner)
 register a host1x driver

Parameters

struct host1x_driver *driver
 host1x driver

struct module *owner
 owner module

Description

Drivers for host1x logical devices call this function to register a driver with the infrastructure. Note that since these drive logical devices, the registration of the driver actually triggers the logical device creation. A logical device will be created for each host1x instance.

void **host1x_driver_unregister**(struct *host1x_driver* *driver)
 unregister a host1x driver

Parameters

struct host1x_driver *driver
 host1x driver

Description

Unbinds the driver from each of the host1x logical devices that it is bound to, effectively removing the subsystem devices that they represent.

void **__host1x_client_init**(struct *host1x_client* *client, struct lock_class_key *key)
 initialize a host1x client

Parameters

struct host1x_client *client
 host1x client

struct lock_class_key *key
 lock class key for the client-specific mutex

void **host1x_client_exit**(struct *host1x_client* *client)
 uninitialize a host1x client

Parameters

struct host1x_client *client
host1x client

int **__host1x_client_register**(struct *host1x_client* *client)
register a host1x client

Parameters

struct host1x_client *client
host1x client

Description

Registers a host1x client with each host1x controller instance. Note that each client will only match their parent host1x controller and will only be associated with that instance. Once all clients have been registered with their parent host1x controller, the infrastructure will set up the logical device and call *host1x_device_init()*, which will in turn call each client's *host1x_client_ops.init* implementation.

void **host1x_client_unregister**(struct *host1x_client* *client)
unregister a host1x client

Parameters

struct host1x_client *client
host1x client

Description

Removes a host1x client from its host1x controller instance. If a logical device has already been initialized, it will be torn down.

Host1x Syncpoint Reference

struct host1x_syncpt ***host1x_syncpt_alloc**(struct host1x *host, unsigned long flags, const char *name)
allocate a syncpoint

Parameters

struct host1x *host
host1x device data

unsigned long flags
bitfield of HOST1X_SYNCPT_* flags

const char *name
name for the syncpoint for use in debug prints

Description

Allocates a hardware syncpoint for the caller's use. The caller then has the sole authority to mutate the syncpoint's value until it is freed again.

If no free syncpoints are available, or a NULL name was specified, returns NULL.

u32 **host1x_syncpt_id**(struct host1x_syncpt *sp)
retrieve syncpoint ID

Parameters

struct host1x_syncpt *sp
host1x syncpoint

Description

Given a pointer to a struct host1x_syncpt, retrieves its ID. This ID is often used as a value to program into registers that control how hardware blocks interact with syncpoints.

u32 **host1x_syncpt_incr_max**(struct host1x_syncpt *sp, u32 incrs)
update the value sent to hardware

Parameters

struct host1x_syncpt *sp
host1x syncpoint

u32 incrs
number of increments

int **host1x_syncpt_incr**(struct host1x_syncpt *sp)
increment syncpoint value from CPU, updating cache

Parameters

struct host1x_syncpt *sp
host1x syncpoint

int **host1x_syncpt_wait**(struct host1x_syncpt *sp, u32 thresh, long timeout, u32 *value)
wait for a syncpoint to reach a given value

Parameters

struct host1x_syncpt *sp
host1x syncpoint

u32 thresh
threshold

long timeout
maximum time to wait for the syncpoint to reach the given value

u32 *value
return location for the syncpoint value

struct host1x_syncpt ***host1x_syncpt_request**(struct *host1x_client* *client, unsigned long flags)
request a syncpoint

Parameters

struct host1x_client *client
client requesting the syncpoint

unsigned long flags
flags

Description

host1x client drivers can use this function to allocate a syncpoint for subsequent use. A syncpoint returned by this function will be reserved for use by the client exclusively. When no longer using a syncpoint, a host1x client driver needs to release it using [*host1x_syncpt_put\(\)*](#).

```
void host1x_syncpt_put(struct host1x_syncpt *sp)
    free a requested syncpoint
```

Parameters

```
struct host1x_syncpt *sp
    host1x syncpoint
```

Description

Release a syncpoint previously allocated using [*host1x_syncpt_request\(\)*](#). A host1x client driver should call this when the syncpoint is no longer in use.

```
u32 host1x_syncpt_read_max(struct host1x_syncpt *sp)
    read maximum syncpoint value
```

Parameters

```
struct host1x_syncpt *sp
    host1x syncpoint
```

Description

The maximum syncpoint value indicates how many operations there are in queue, either in channel or in a software thread.

```
u32 host1x_syncpt_read_min(struct host1x_syncpt *sp)
    read minimum syncpoint value
```

Parameters

```
struct host1x_syncpt *sp
    host1x syncpoint
```

Description

The minimum syncpoint value is a shadow of the current sync point value in hardware.

```
u32 host1x_syncpt_read(struct host1x_syncpt *sp)
    read the current syncpoint value
```

Parameters

```
struct host1x_syncpt *sp
    host1x syncpoint
```

```
struct host1x_syncpt *host1x_syncpt_get_by_id(struct host1x *host, unsigned int id)
    obtain a syncpoint by ID
```

Parameters

```
struct host1x *host
    host1x controller
```

```
unsigned int id
    syncpoint ID
```

`struct host1x_syncpt *host1x_syncpt_get_by_id_noref(struct host1x *host, unsigned int id)`
obtain a syncpoint by ID but don't increase the refcount.

Parameters

struct host1x *host
host1x controller

unsigned int id
syncpoint ID

`struct host1x_syncpt *host1x_syncpt_get(struct host1x_syncpt *sp)`
increment syncpoint refcount

Parameters

struct host1x_syncpt *sp
syncpoint

`struct host1x_syncpt_base *host1x_syncpt_get_base(struct host1x_syncpt *sp)`
obtain the wait base associated with a syncpoint

Parameters

struct host1x_syncpt *sp
host1x syncpoint

`u32 host1x_syncpt_base_id(struct host1x_syncpt_base *base)`
retrieve the ID of a syncpoint wait base

Parameters

struct host1x_syncpt_base *base
host1x syncpoint wait base

`void host1x_syncpt_release_vblank_reservation(struct host1x_client *client, u32 syncpt_id)`

Make VBLANK syncpoint available for allocation

Parameters

struct host1x_client *client
host1x bus client

u32 syncpt_id
syncpoint ID to make available

Description

Makes VBLANK<i>i

10.6.2 KMS driver

The display hardware has remained mostly backwards compatible over the various Tegra SoC generations, up until Tegra186 which introduces several changes that make it difficult to support with a parameterized driver.

Display Controllers

Tegra SoCs have two display controllers, each of which can be associated with zero or more outputs. Outputs can also share a single display controller, but only if they run with compatible display timings. Two display controllers can also share a single framebuffer, allowing cloned configurations even if modes on two outputs don't match. A display controller is modelled as a CRTIC in KMS terms.

On Tegra186, the number of display controllers has been increased to three. A display controller can no longer drive all of the outputs. While two of these controllers can drive both DSI outputs and both SOR outputs, the third cannot drive any DSI.

Windows

A display controller controls a set of windows that can be used to composite multiple buffers onto the screen. While it is possible to assign arbitrary Z ordering to individual windows (by programming the corresponding blending registers), this is currently not supported by the driver. Instead, it will assume a fixed Z ordering of the windows (window A is the root window, that is, the lowest, while windows B and C are overlaid on top of window A). The overlay windows support multiple pixel formats and can automatically convert from YUV to RGB at scanout time. This makes them useful for displaying video content. In KMS, each window is modelled as a plane. Each display controller has a hardware cursor that is exposed as a cursor plane.

Outputs

The type and number of supported outputs varies between Tegra SoC generations. All generations support at least HDMI. While earlier generations supported the very simple RGB interfaces (one per display controller), recent generations no longer do and instead provide standard interfaces such as DSI and eDP/DP.

Outputs are modelled as a composite encoder/connector pair.

RGB/LVDS

This interface is no longer available since Tegra124. It has been replaced by the more standard DSI and eDP interfaces.

HDMI

HDMI is supported on all Tegra SoCs. Starting with Tegra210, HDMI is provided by the versatile SOR output, which supports eDP, DP and HDMI. The SOR is able to support HDMI 2.0, though support for this is currently not merged.

DSI

Although Tegra has supported DSI since Tegra30, the controller has changed in several ways in Tegra114. Since none of the publicly available development boards prior to Dalmore (Tegra114) have made use of DSI, only Tegra114 and later are supported by the `drm/tegra` driver.

eDP/DP

eDP was first introduced in Tegra124 where it was used to drive the display panel for notebook form factors. Tegra210 added support for full DisplayPort support, though this is currently not implemented in the `drm/tegra` driver.

10.6.3 Userspace Interface

The userspace interface provided by `drm/tegra` allows applications to create GEM buffers, access and control syncpoints as well as submit command streams to `host1x`.

GEM Buffers

The `DRM_IOCTL_TEGRA_GEM_CREATE` IOCTL is used to create a GEM buffer object with Tegra-specific flags. This is useful for buffers that should be tiled, or that are to be scanned out upside down (useful for 3D content).

After a GEM buffer object has been created, its memory can be mapped by an application using the `mmap` offset returned by the `DRM_IOCTL_TEGRA_GEM_MMAP` IOCTL.

Syncpoints

The current value of a syncpoint can be obtained by executing the `DRM_IOCTL_TEGRA_SYNCPT_READ` IOCTL. Incrementing the syncpoint is achieved using the `DRM_IOCTL_TEGRA_SYNCPT_INCR` IOCTL.

Userspace can also request blocking on a syncpoint. To do so, it needs to execute the `DRM_IOCTL_TEGRA_SYNCPT_WAIT` IOCTL, specifying the value of the syncpoint to wait for. The kernel will release the application when the syncpoint reaches that value or after a specified timeout.

Command Stream Submission

Before an application can submit command streams to host1x it needs to open a channel to an engine using the `DRM_IOCTL_TEGRA_OPEN_CHANNEL` IOCTL. Client IDs are used to identify the target of the channel. When a channel is no longer needed, it can be closed using the `DRM_IOCTL_TEGRA_CLOSE_CHANNEL` IOCTL. To retrieve the syncpoint associated with a channel, an application can use the `DRM_IOCTL_TEGRA_GET_SYNCPT`.

After opening a channel, submitting command streams is easy. The application writes commands into the memory backing a GEM buffer object and passes these to the `DRM_IOCTL_TEGRA_SUBMIT` IOCTL along with various other parameters, such as the syncpoints or relocations used in the job submission.

10.7 drm/tve200 Faraday TV Encoder 200

The Faraday TV Encoder TVE200 is also known as the Gemini TV Interface Controller (TVC) and is found in the Gemini Chipset from Storlink Semiconductor (later Storm Semiconductor, later Cortina Systems) but also in the Grain Media GM8180 chipset. On the Gemini the module is connected to 8 data lines and a single clock line, comprising an 8-bit BT.656 interface.

This is a very basic YUV display driver. The datasheet specifies that it supports the ITU BT.656 standard. It requires a 27 MHz clock which is the hallmark of any TV encoder supporting both PAL and NTSC.

This driver exposes a standard KMS interface for this TV encoder.

10.8 drm/v3d Broadcom V3D Graphics Driver

This driver supports the Broadcom V3D 3.3 and 4.1 OpenGL ES GPUs. For V3D 2.x support, see the VC4 driver.

The V3D GPU includes a tiled render (composed of a bin and render pipelines), the TFU (texture formatting unit), and the CSD (compute shader dispatch).

10.8.1 GPU buffer object (BO) management

Compared to VC4 (V3D 2.x), V3D 3.3 introduces an MMU between the GPU and the bus, allowing us to use shmem objects for our storage instead of CMA.

Physically contiguous objects may still be imported to V3D, but the driver doesn't allocate physically contiguous objects on its own. Display engines requiring physically contiguous allocations should look into Mesa's "renderonly" support (as used by the Mesa pl111 driver) for an example of how to integrate with V3D.

Long term, we should support evicting pages from the MMU when under memory pressure (thus the `v3d_bo_get_pages()` refcounting), but that's not a high priority since our systems tend to not have swap.

Address space management

The V3D 3.x hardware (compared to VC4) now includes an MMU. It has a single level of page tables for the V3D's 4GB address space to map to AXI bus addresses, thus it could need up to 4MB of physically contiguous memory to store the PTEs.

Because the 4MB of contiguous memory for page tables is precious, and switching between them is expensive, we load all BOs into the same 4GB address space.

To protect clients from each other, we should use the GMP to quickly mask out (at 128kb granularity) what pages are available to each client. This is not yet implemented.

GPU Scheduling

The shared DRM GPU scheduler is used to coordinate submitting jobs to the hardware. Each DRM fd (roughly a client process) gets its own scheduler entity, which will process jobs in order. The GPU scheduler will round-robin between clients to submit the next job.

For simplicity, and in order to keep latency low for interactive jobs when bulk background jobs are queued up, we submit a new job to the HW only when it has completed the last one, instead of filling up the CT[01]Q FIFOs with jobs. Similarly, we use `drm_sched_job_add_dependency()` to manage the dependency between bin and render, instead of having the clients submit jobs using the HW's semaphores to interlock between them.

10.8.2 Interrupts

When we take a bin, render, TFU done, or CSD done interrupt, we need to signal the fence for that job so that the scheduler can queue up the next one and unblock any waiters.

When we take the binner out of memory interrupt, we need to allocate some new memory and pass it to the binner so that the current job can make progress.

10.9 drm/vc4 Broadcom VC4 Graphics Driver

The Broadcom VideoCore 4 (present in the Raspberry Pi) contains a OpenGL ES 2.0-compatible 3D engine called V3D, and a highly configurable display output pipeline that supports HDMI, DSI, DPI, and Composite TV output.

The 3D engine also has an interface for submitting arbitrary compute shader-style jobs using the same shader processor as is used for vertex and fragment shaders in GLES 2.0. However, given that the hardware isn't able to expose any standard interfaces like OpenGL compute shaders or OpenCL, it isn't supported by this driver.

10.9.1 Display Hardware Handling

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

Pixel Valve (DRM CRTC)

In VC4, the Pixel Valve is what most closely corresponds to the DRM's concept of a CRTC. The PV generates video timings from the encoder's clock plus its configuration. It pulls scaled pixels from the HVS at that timing, and feeds it to the encoder.

However, the DRM CRTC also collects the configuration of all the DRM planes attached to it. As a result, the CRTC is also responsible for writing the display list for the HVS channel that the CRTC will use.

The 2835 has 3 different pixel valves. pv0 in the audio power domain feeds DSI0 or DPI, while pv1 feeds DS1 or SMI. pv2 in the image domain can feed either HDMI or the SDTV controller. The pixel valve chooses from the CPRMAN clocks (HSM for HDMI, VEC for SDTV, etc.) according to which output type is chosen in the mux.

For power management, the pixel valve's registers are all clocked by the AXI clock, while the timings and FIFOs make use of the output-specific clock. Since the encoders also directly consume the CPRMAN clocks, and know what timings they need, they are the ones that set the clock.

HVS

The Hardware Video Scaler (HVS) is the piece of hardware that does translation, scaling, colorspace conversion, and compositing of pixels stored in framebuffers into a FIFO of pixels going out to the Pixel Valve (CRTC). It operates at the system clock rate (the system audio clock gate, specifically), which is much higher than the pixel clock rate.

There is a single global HVS, with multiple output FIFOs that can be consumed by the PVs. This file just manages the resources for the HVS, while the `vc4_crtc.c` code actually drives HVS setup for each CRTC.

HVS planes

Each DRM plane is a layer of pixels being scanned out by the HVS.

At atomic modeset check time, we compute the HVS display element state that would be necessary for displaying the plane (giving us a chance to figure out if a plane configuration is invalid), then at atomic flush time the CRTC will ask us to write our element state into the region of the HVS that it has allocated for us.

HDMI encoder

The HDMI core has a state machine and a PHY. On BCM2835, most of the unit operates off of the HSM clock from CPRMAN. It also internally uses the PLLH_PIX clock for the PHY.

HDMI infoframes are kept within a small packet ram, where each packet can be individually enabled for including in a frame.

HDMI audio is implemented entirely within the HDMI IP block. A register in the HDMI encoder takes SPDIF frames from the DMA engine and transfers them over an internal MAI (multi-channel audio interconnect) bus to the encoder side for insertion into the video blank regions.

The driver's HDMI encoder does not yet support power management. The HDMI encoder's power domain and the HSM/pixel clocks are kept continuously running, and only the HDMI logic and packet ram are powered off/on at disable/enable time.

The driver does not yet support CEC control, though the HDMI encoder block has CEC support.

DSI encoder

BCM2835 contains two DSI modules, DSI0 and DSI1. DSI0 is a single-lane DSI controller, while DSI1 is a more modern 4-lane DSI controller.

Most Raspberry Pi boards expose DSI1 as their "DISPLAY" connector, while the compute module brings both DSI0 and DSI1 out.

This driver has been tested for DSI1 video-mode display only currently, with most of the information necessary for DSI0 hopefully present.

DPI encoder

The VC4 DPI hardware supports MIPI DPI type 4 and Nokia ViSSI signals. On BCM2835, these can be routed out to GPIO0-27 with the ALT2 function.

VEC (Composite TV out) encoder

The VEC encoder generates PAL or NTSC composite video output.

TV mode selection is done by an atomic property on the encoder, because a `drm_mode_modeinfo` is insufficient to distinguish between PAL and PAL-M or NTSC and NTSC-J.

10.9.2 KUnit Tests

The VC4 Driver uses KUnit to perform driver-specific unit and integration tests.

These tests are using a mock driver and can be ran using the command below, on either arm or arm64 architectures,

```
$ ./tools/testing/kunit/kunit.py run \
    --kunitconfig=drivers/gpu/drm/vc4/tests/.kunitconfig \
    --cross_compile aarch64-linux-gnu- --arch arm64
```

Parts of the driver that are currently covered by tests are:

- The HVS to PixelValve dynamic FIFO assignment, for the BCM2835-7 and BCM2711.

10.9.3 Memory Management and 3D Command Submission

This section covers the GEM implementation in the vc4 driver.

GPU buffer object (BO) management

The VC4 GPU architecture (both scanout and rendering) has direct access to system memory with no MMU in between. To support it, we use the GEM DMA helper functions to allocate contiguous ranges of physical memory for our BOs.

Since the DMA allocator is very slow, we keep a cache of recently freed BOs around so that the kernel's allocation of objects for 3D rendering can return quickly.

V3D binner command list (BCL) validation

Since the VC4 has no IOMMU between it and system memory, a user with access to execute command lists could escalate privilege by overwriting system memory (drawing to it as a framebuffer) or reading system memory it shouldn't (reading it as a vertex buffer or index buffer)

We validate binner command lists to ensure that all accesses are within the bounds of the GEM objects referenced by the submitted job. It explicitly whitelists packets, and looks at the offsets in any address fields to make sure they're contained within the BOs they reference.

Note that because CL validation is already reading the user-submitted CL and writing the validated copy out to the memory that the GPU will actually read, this is also where GEM relocation processing (turning BO references into actual addresses for the GPU to use) happens.

V3D render command list (RCL) generation

In the V3D hardware, render command lists are what load and store tiles of a framebuffer and optionally call out to binner-generated command lists to do the 3D drawing for that tile.

In the VC4 driver, render command list generation is performed by the kernel instead of userspace. We do this because validating a user-submitted command list is hard to get right and has high CPU overhead, while the number of valid configurations for render command lists is actually fairly low.

Shader validator for VC4

Since the VC4 has no IOMMU between it and system memory, a user with access to execute shaders could escalate privilege by overwriting system memory (using the VPM write address register in the general-purpose DMA mode) or reading system memory it shouldn't (reading it as a texture, uniform data, or direct-addressed TMU lookup).

The shader validator walks over a shader's BO, ensuring that its accesses are appropriately bounded, and recording where texture accesses are made so that we can do relocations for them in the uniform stream.

Shader BO are immutable for their lifetimes (enforced by not allowing mmmaps, GEM prime export, or rendering to from a CL), so this validation is only performed at BO creation time.

V3D Interrupts

We have an interrupt status register (V3D_INTCTL) which reports interrupts, and where writing 1 bits clears those interrupts. There are also a pair of interrupt registers (V3D_INTENA/V3D_INTDIS) where writing a 1 to their bits enables or disables that specific interrupt, and 0s written are ignored (reading either one returns the set of enabled interrupts).

When we take a binning flush done interrupt, we need to submit the next frame for binning and move the finished frame to the render thread.

When we take a render frame interrupt, we need to wake the processes waiting for some frame to be done, and get the next frame submitted ASAP (so the hardware doesn't sit idle when there's work to do).

When we take the binner out of memory interrupt, we need to allocate some new memory and pass it to the binner so that the current job can make progress.

10.10 drm/vkms Virtual Kernel Modesetting

VKMS is a software-only model of a KMS driver that is useful for testing and for running X (or similar) on headless machines. VKMS aims to enable a virtual display with no need of a hardware display capability, releasing the GPU in DRM API tests.

10.10.1 Setup

The VKMS driver can be setup with the following steps:

To check if VKMS is loaded, run:

```
lsmod | grep vkms
```

This should list the VKMS driver. If no output is obtained, then you need to enable and/or load the VKMS driver. Ensure that the VKMS driver has been set as a loadable module in your kernel config file. Do:

```
make nconfig
```

```
Go to `Device Drivers> Graphics support`
```

```
Enable `Virtual KMS (EXPERIMENTAL)`
```

Compile and build the kernel for the changes to get reflected. Now, to load the driver, use:

```
sudo modprobe vkms
```

On running the lsmod command now, the VKMS driver will appear listed. You can also observe the driver being loaded in the dmesg logs.

The VKMS driver has optional features to simulate different kinds of hardware, which are exposed as module options. You can use the *modinfo* command to see the module options for vkms:

```
modinfo vkms
```

Module options are helpful when testing, and enabling modules can be done while loading vkms. For example, to load vkms with cursor enabled, use:

```
sudo modprobe vkms enable_cursor=1
```

To disable the driver, use

```
sudo modprobe -r vkms
```

10.10.2 Testing With IGT

The IGT GPU Tools is a test suite used specifically for debugging and development of the DRM drivers. The IGT Tools can be installed from [here](#).

The tests need to be run without a compositor, so you need to switch to text only mode. You can do this by:

```
sudo systemctl isolate multi-user.target
```

To return to graphical mode, do:

```
sudo systemctl isolate graphical.target
```

Once you are in text only mode, you can run tests using the `--device` switch or `IGT_DEVICE` variable to specify the device filter for the driver we want to test. `IGT_DEVICE` can also be used with the `run-test.sh` script to run the tests for a specific driver:

```
sudo ./build/tests/<name of test> --device "sys:/sys/devices/platform/vkms"
sudo IGT_DEVICE="sys:/sys/devices/platform/vkms" ./build/tests/<name of test>
sudo IGT_DEVICE="sys:/sys/devices/platform/vkms" ./scripts/run-tests.sh -t
↳<name of test>
```

For example, to test the functionality of the writeback library, we can run the `kms_writeback` test:

```
sudo ./build/tests/kms_writeback --device "sys:/sys/devices/platform/vkms"
sudo IGT_DEVICE="sys:/sys/devices/platform/vkms" ./build/tests/kms_writeback
sudo IGT_DEVICE="sys:/sys/devices/platform/vkms" ./scripts/run-tests.sh -t kms_
↳writeback
```

You can also run subtests if you do not want to run the entire test:

```
sudo ./build/tests/kms_flip --run-subtest basic-plain-flip --device "sys:/sys/
↳devices/platform/vkms"
sudo IGT_DEVICE="sys:/sys/devices/platform/vkms" ./build/tests/kms_flip --run-
↳subtest basic-plain-flip
```


10.10.3 TODO

If you want to do any of the items listed below, please share your interest with VKMS maintainers.

IGT better support

Debugging:

- kms_plane: some test cases are failing due to timeout on capturing CRC;

Virtual hardware (vblank-less) mode:

- VKMS already has support for vblanks simulated via hrtimers, which can be tested with kms_flip test; in some way, we can say that VKMS already mimics the real hardware vblank. However, we also have virtual hardware that does not support vblank interrupt and completes page_flip events right away; in this case, compositor developers may end up creating a busy loop on virtual hardware. It would be useful to support Virtual Hardware behavior in VKMS because this can help compositor developers to test their features in multiple scenarios.

Add Plane Features

There's lots of plane features we could add support for:

- Add background color KMS property[Good to get started].
- Scaling.
- Additional buffer formats, especially YUV formats for video like NV12. Low/high bpp RGB formats would also be interesting.
- Async updates (currently only possible on cursor plane using the legacy cursor api).

For all of these, we also want to review the igt test coverage and make sure all relevant igt testcases work on vkms. They are good options for internship project.

Runtime Configuration

We want to be able to reconfigure vkms instance without having to reload the module. Use/Test-cases:

- Hotplug/hotremove connectors on the fly (to be able to test DP MST handling of compositors).
- Configure planes/crtcs/connectors (we'd need some code to have more than 1 of them first).
- Change output configuration: Plug/unplug screens, change EDID, allow changing the refresh rate.

The currently proposed solution is to expose vkms configuration through configs. All existing module options should be supported through configs too.

Writeback support

- The writeback and CRC capture operations share the use of `composer_enabled` boolean to ensure vblanks. Probably, when these operations work together, `composer_enabled` needs to refcounting the composer state to proper work. [Good to get started]
- Add support for cloned writeback outputs and related test cases using a cloned output in the IGT `kms_writeback`.
- As a v4l device. This is useful for debugging compositors on special vkms configurations, so that developers see what's really going on.

Output Features

- Variable refresh rate/freesync support. This probably needs prime buffer sharing support, so that we can use vgem fences to simulate rendering in testing. Also needs support to specify the EDID.
- Add support for link status, so that compositors can validate their runtime fallbacks when e.g. a Display Port link goes bad.

CRC API Improvements

- Optimize CRC computation `compute_crc()` and plane blending `blend()`

Atomic Check using eBPF

Atomic drivers have lots of restrictions which are not exposed to userspace in any explicit form through e.g. possible property values. Userspace can only inquiry about these limits through the atomic IOCTL, possibly using the `TEST_ONLY` flag. Trying to add configurable code for all these limits, to allow compositors to be tested against them, would be rather futile exercise. Instead we could add support for eBPF to validate any kind of atomic state, and implement a library of different restrictions.

This needs a bunch of features (plane compositing, multiple outputs, ...) enabled already to make sense.

10.11 drm/bridge/dw-hdmi Synopsys DesignWare HDMI Controller

10.11.1 Synopsys DesignWare HDMI Controller

This section covers everything related to the Synopsys DesignWare HDMI Controller implemented as a DRM bridge.

Supported Input Formats and Encodings

Depending on the Hardware configuration of the Controller IP, it supports a subset of the following input formats and encodings on its internal 48bit bus.

Format Name	Format Code	Encodings
RGB 4:4:4 8bit	MEDIA_BUS_FMT_RGB888_1X24	V4L2_YCBCR_ENC_DEFAULT
RGB 4:4:4 10bits	MEDIA_BUS_FMT_RGB101010_1X30	V4L2_YCBCR_ENC_DEFAULT
RGB 4:4:4 12bits	MEDIA_BUS_FMT_RGB121212_1X36	V4L2_YCBCR_ENC_DEFAULT
RGB 4:4:4 16bits	MEDIA_BUS_FMT_RGB161616_1X48	V4L2_YCBCR_ENC_DEFAULT
YCbCr 4:4:4 8bit	MEDIA_BUS_FMT_YUV8_1X24	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709 or
		V4L2_YCBCR_ENC_XV601 or
		V4L2_YCBCR_ENC_XV709
YCbCr 4:4:4 10bits	MEDIA_BUS_FMT_YUV10_1X30	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709 or
		V4L2_YCBCR_ENC_XV601 or
		V4L2_YCBCR_ENC_XV709
YCbCr 4:4:4 12bits	MEDIA_BUS_FMT_YUV12_1X36	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709 or
		V4L2_YCBCR_ENC_XV601 or
		V4L2_YCBCR_ENC_XV709
YCbCr 4:4:4 16bits	MEDIA_BUS_FMT_YUV16_1X48	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709 or
		V4L2_YCBCR_ENC_XV601 or
		V4L2_YCBCR_ENC_XV709
YCbCr 4:2:2 8bit	MEDIA_BUS_FMT_UYVY8_1X16	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709
YCbCr 4:2:2 10bits	MEDIA_BUS_FMT_UYVY10_1X20	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709
YCbCr 4:2:2 12bits	MEDIA_BUS_FMT_UYVY12_1X24	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709
YCbCr 4:2:0 8bit	MEDIA_BUS_FMT_UYYVYY8_0_5	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709
YCbCr 4:2:0 10bits	MEDIA_BUS_FMT_UYYVYY10_0_5	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709
YCbCr 4:2:0 12bits	MEDIA_BUS_FMT_UYYVYY12_0_5	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709
YCbCr 4:2:0 16bits	MEDIA_BUS_FMT_UYYVYY16_0_5	V4L2_YCBCR_ENC_601 or
		V4L2_YCBCR_ENC_709

10.12 drm/xen-front Xen para-virtualized frontend driver

This frontend driver implements Xen para-virtualized display according to the display protocol described at `include/xen/interface/io/displif.h`

10.12.1 Driver modes of operation in terms of display buffers used

Depending on the requirements for the para-virtualized environment, namely requirements dictated by the accompanying DRM/(v)GPU drivers running in both host and guest environments, display buffers can be allocated by either frontend driver or backend.

Buffers allocated by the frontend driver

In this mode of operation driver allocates buffers from system memory.

Note! If used with accompanying DRM/(v)GPU drivers this mode of operation may require IOMMU support on the platform, so accompanying DRM/vGPU hardware can still reach display buffer memory while importing PRIME buffers from the frontend driver.

Buffers allocated by the backend

This mode of operation is run-time configured via guest domain configuration through XenStore entries.

For systems which do not provide IOMMU support, but having specific requirements for display buffers it is possible to allocate such buffers at backend side and share those with the frontend. For example, if host domain is 1:1 mapped and has DRM/GPU hardware expecting physically contiguous memory, this allows implementing zero-copying use-cases.

Note, while using this scenario the following should be considered:

1. If guest domain dies then pages/grants received from the backend cannot be claimed back
2. Misbehaving guest may send too many requests to the backend exhausting its grant references and memory (consider this from security POV)

10.12.2 Driver limitations

1. Only primary plane without additional properties is supported.
2. Only one video mode per connector supported which is configured via XenStore.
3. All CRTC's operate at fixed frequency of 60Hz.

10.13 Arm Framebuffer Compression (AFBC)

AFBC is a proprietary lossless image compression protocol and format. It provides fine-grained random access and minimizes the amount of data transferred between IP blocks.

AFBC can be enabled on drivers which support it via use of the AFBC format modifiers defined in `drm_fourcc.h`. See `DRM_FORMAT_MOD_ARM_AFBC(*)`.

All users of the AFBC modifiers must follow the usage guidelines laid out in this document, to ensure compatibility across different AFBC producers and consumers.

10.13.1 Components and Ordering

AFBC streams can contain several components - where a component corresponds to a color channel (i.e. R, G, B, X, A, Y, Cb, Cr). The assignment of input/output color channels must be consistent between the encoder and the decoder for correct operation, otherwise the consumer will interpret the decoded data incorrectly.

Furthermore, when the lossless colorspace transform is used (`AFBC_FORMAT_MOD_YTR`, which should be enabled for RGB buffers for maximum compression efficiency), the component order must be:

- Component 0: R
- Component 1: G
- Component 2: B

The component ordering is communicated via the fourcc code in the `fourcc:modifier` pair. In general, component '0' is considered to reside in the least-significant bits of the corresponding linear format. For example, `COMP(bits)`:

- `DRM_FORMAT_ABGR8888`
 - Component 0: R(8)
 - Component 1: G(8)
 - Component 2: B(8)
 - Component 3: A(8)
- `DRM_FORMAT_BGR888`
 - Component 0: R(8)
 - Component 1: G(8)
 - Component 2: B(8)
- `DRM_FORMAT_YUYV`
 - Component 0: Y(8)
 - Component 1: Cb(8, 2x1 subsampled)
 - Component 2: Cr(8, 2x1 subsampled)

In AFBC, 'X' components are not treated any differently from any other component. Therefore, an AFBC buffer with fourcc `DRM_FORMAT_XBGR8888` encodes with 4 components, like so:

- `DRM_FORMAT_XBGR8888`
 - Component 0: R(8)
 - Component 1: G(8)
 - Component 2: B(8)
 - Component 3: X(8)

Please note, however, that the inclusion of a “wasted” ‘X’ channel is bad for compression efficiency, and so it’s recommended to avoid formats containing ‘X’ bits. If a fourth component is required/expected by the encoder/decoder, then it is recommended to instead use an equivalent format with alpha, setting all alpha bits to ‘1’. If there is no requirement for a fourth component, then a format which doesn’t include alpha can be used, e.g. `DRM_FORMAT_BGR888`.

10.13.2 Number of Planes

Formats which are typically multi-planar in linear layouts (e.g. YUV 420), can be encoded into one, or multiple, AFBC planes. As with component order, the encoder and decoder must agree about the number of planes in order to correctly decode the buffer. The fourcc code is used to determine the number of encoded planes in an AFBC buffer, matching the number of planes for the linear (unmodified) format. Within each plane, the component ordering also follows the fourcc code:

For example:

- `DRM_FORMAT_YUYV`: `nplanes = 1`
 - Plane 0:
 - * Component 0: Y(8)
 - * Component 1: Cb(8, 2x1 subsampled)
 - * Component 2: Cr(8, 2x1 subsampled)
- `DRM_FORMAT_NV12`: `nplanes = 2`
 - Plane 0:
 - * Component 0: Y(8)
 - Plane 1:
 - * Component 0: Cb(8, 2x1 subsampled)
 - * Component 1: Cr(8, 2x1 subsampled)

10.13.3 Cross-device interoperability

For maximum compatibility across devices, the table below defines canonical formats for use between AFBC-enabled devices. Formats which are listed here must be used exactly as specified when using the AFBC modifiers. Formats which are not listed should be avoided.

Table 1: AFBC formats

Fourcc code	Description	Planes/Components
DRM_FORMAT_ABGR2101010	10-bit per component RGB, with 2-bit alpha	Plane 0: 4 components <ul style="list-style-type: none">Component 0: R(10)Component 1: G(10)Component 2: B(10)Component 3: A(2)
DRM_FORMAT_ABGR8888	8-bit per component RGB, with 8-bit alpha	Plane 0: 4 components <ul style="list-style-type: none">Component 0: R(8)Component 1: G(8)Component 2: B(8)Component 3: A(8)
DRM_FORMAT_BGR888	8-bit per component RGB	Plane 0: 3 components <ul style="list-style-type: none">Component 0: R(8)Component 1: G(8)Component 2: B(8)
DRM_FORMAT_BGR565	5/6-bit per component RGB	Plane 0: 3 components <ul style="list-style-type: none">Component 0: R(5)Component 1: G(6)Component 2: B(5)
DRM_FORMAT_ABGR1555	5-bit per component RGB, with 1-bit alpha	Plane 0: 4 components <ul style="list-style-type: none">Component 0: R(5)Component 1: G(5)Component 2: B(5)Component 3: A(1)
DRM_FORMAT_VUY888	8-bit per component YCbCr 444, single plane	Plane 0: 3 components <ul style="list-style-type: none">Component 0: Y(8)Component 1: Cb(8)Component 2: Cr(8)
DRM_FORMAT_VUY101010	10-bit per component YCbCr 444, single plane	Plane 0: 3 components <ul style="list-style-type: none">Component 0: Y(10)Component 1: Cb(10)Component 2: Cr(10)
DRM_FORMAT_YUYV	8-bit per component YCbCr 422, single plane	Plane 0: 3 components <ul style="list-style-type: none">Component 0: Y(8)Component 1: Cb(8, 2x1 subsampled)Component 2: Cr(8, 2x1 subsampled)

10.13. Arm Framebuffer Compression (AFBC)

- Component 0: Y(8)
- Component 1: Cb(8, 2x1 subsampled)
- Component 2: Cr(8, 2x1 subsampled)

10.14 drm/komeda Arm display driver

The drm/komeda driver supports the Arm display processor D71 and later products, this document gives a brief overview of driver design: how it works and why design it like that.

10.14.1 Overview of D71 like display IPs

From D71, Arm display IP begins to adopt a flexible and modularized architecture. A display pipeline is made up of multiple individual and functional pipeline stages called components, and every component has some specific capabilities that can give the flowed pipeline pixel data a particular processing.

Typical D71 components:

Layer

Layer is the first pipeline stage, which prepares the pixel data for the next stage. It fetches the pixel from memory, decodes it if it's AFBC, rotates the source image, unpacks or converts YUV pixels to the device internal RGB pixels, then adjusts the color_space of pixels if needed.

Scaler

As its name suggests, scaler takes responsibility for scaling, and D71 also supports image enhancements by scaler. The usage of scaler is very flexible and can be connected to layer output for layer scaling, or connected to compositor and scale the whole display frame and then feed the output data into wb_layer which will then write it into memory.

Compositor (compiz)

Compositor blends multiple layers or pixel data flows into one single display frame. its output frame can be fed into post image processor for showing it on the monitor or fed into wb_layer and written to memory at the same time. user can also insert a scaler between compositor and wb_layer to down scale the display frame first and then write to memory.

Writeback Layer (wb_layer)

Writeback layer does the opposite things of Layer, which connects to compiz and writes the composition result to memory.

Post image processor (improc)

Post image processor adjusts frame data like gamma and color space to fit the requirements of the monitor.

Timing controller (timing_ctrlr)

Final stage of display pipeline, Timing controller is not for the pixel handling, but only for controlling the display timing.

Merger

D71 scaler mostly only has the half horizontal input/output capabilities compared with Layer, like if Layer supports 4K input size, the scaler only can support 2K input/output in the same time. To achieve the full frame scaling, D71 introduces Layer Split, which splits the whole image to two half parts and feeds them to two Layers A and B, and does the scaling independently. After scaling the result needs to be fed to merger to merge two part images together, and then output merged result to compiz.

Splitter

Similar to Layer Split, but Splitter is used for writeback, which splits the compiz result to two parts and then feeds them to two scalers.

10.14.2 Possible D71 Pipeline usage

Benefitting from the modularized architecture, D71 pipelines can be easily adjusted to fit different usages. And D71 has two pipelines, which support two types of working mode:

- Dual display mode Two pipelines work independently and separately to drive two display outputs.
- Single display mode Two pipelines work together to drive only one display output.

On this mode, pipeline_B doesn't work independently, but outputs its composition result into pipeline_A, and its pixel timing also derived from pipeline_A.timing_ctrlr. The pipeline_B works just like a "slave" of pipeline_A(master)

Single pipeline data flow

Dual pipeline with Slave enabled

Sub-pipelines for input and output

A complete display pipeline can be easily divided into three sub-pipelines according to the in/out usage.

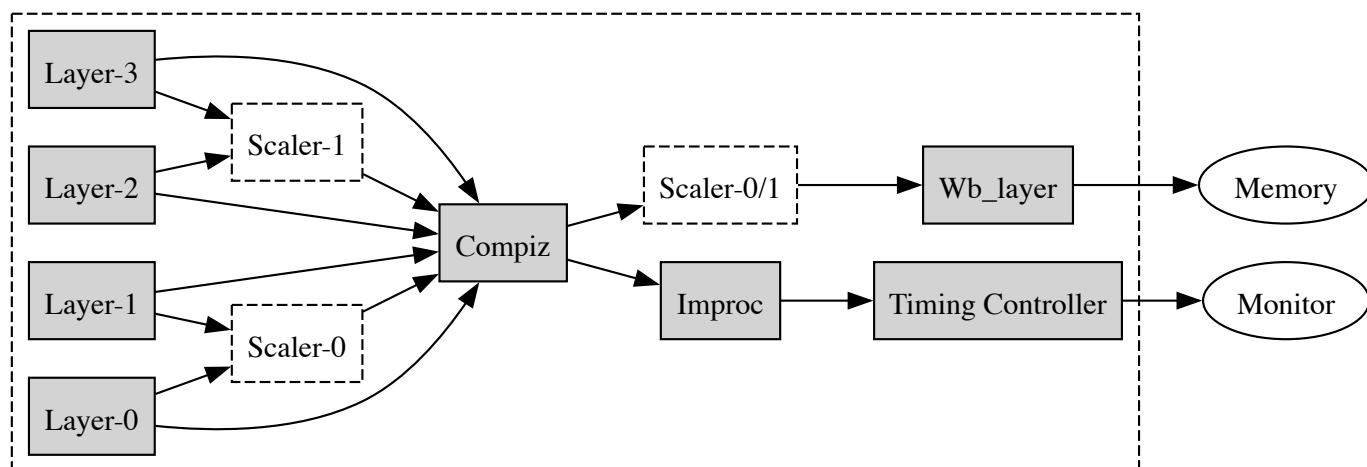


Fig. 1: Single pipeline data flow

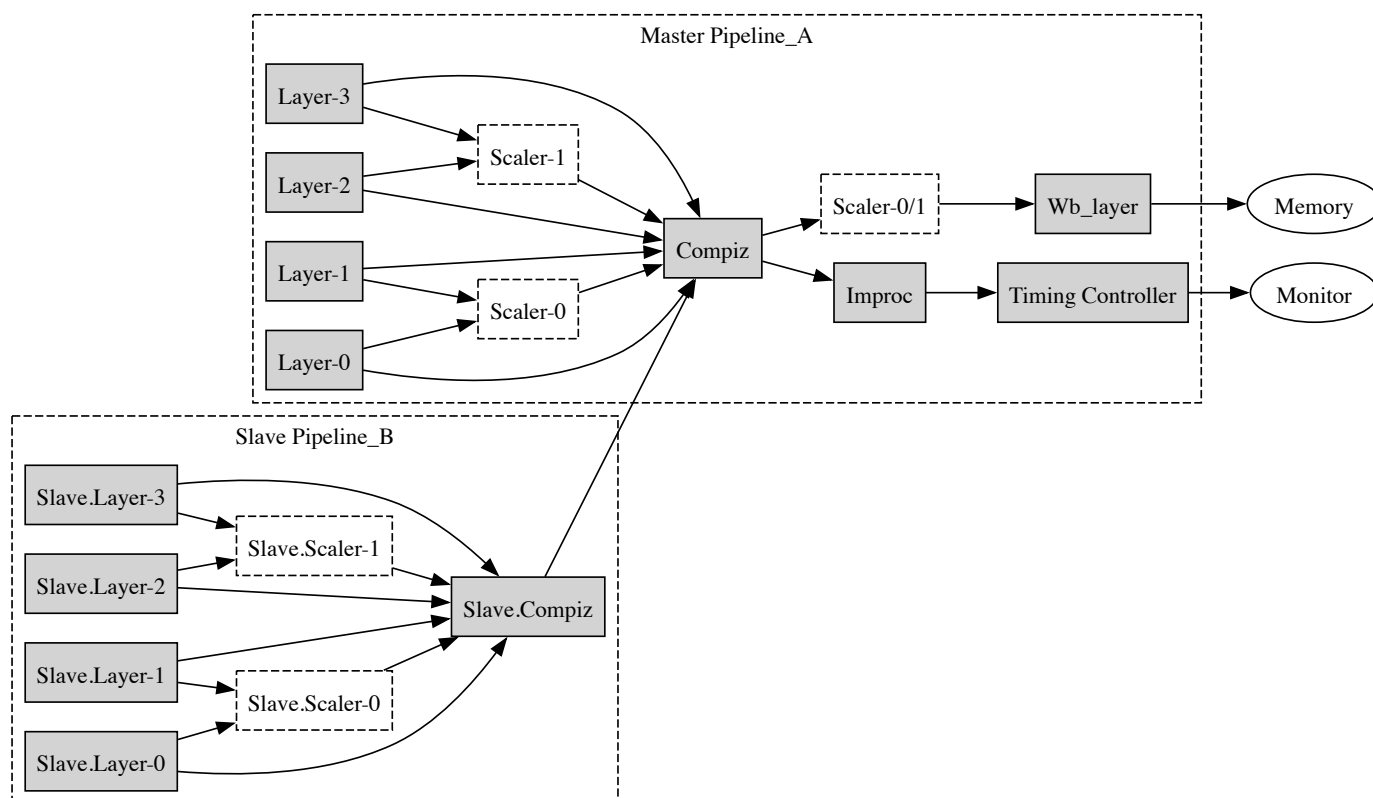


Fig. 2: Slave pipeline enabled data flow

Layer(input) pipeline



Fig. 3: Layer (input) data flow

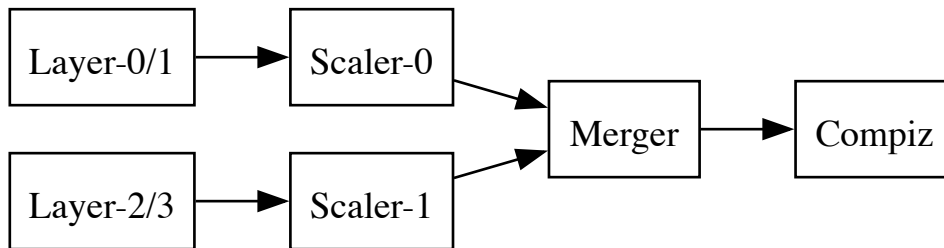


Fig. 4: Layer Split pipeline

Writeback(output) pipeline

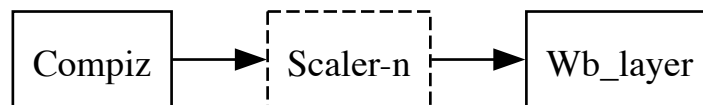


Fig. 5: Writeback(output) data flow

Display output pipeline

In the following section we'll see these three sub-pipelines will be handled by KMS-plane/wb_conn/crtc respectively.

10.14.3 Komeda Resource abstraction

struct komeda_pipeline/component

To fully utilize and easily access/configure the HW, the driver side also uses a similar architecture: Pipeline/Component to describe the HW features and capabilities, and a specific component includes two parts:

- Data flow controlling.
- Specific component capabilities and features.

So the driver defines a common header `struct komeda_component` to describe the data flow control and all specific components are a subclass of this base structure.

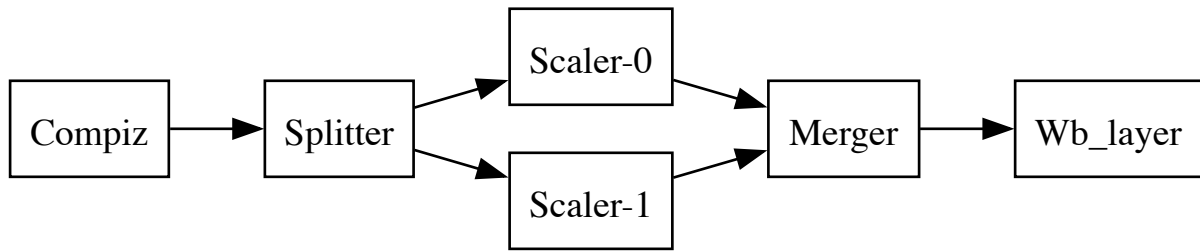


Fig. 6: Writeback(output) Split data flow



Fig. 7: display output data flow

struct **komeda_component**

Definition:

```
struct komeda_component {
    struct drm_private_obj obj;
    struct komeda_pipeline *pipeline;
    char name[32];
    u32 __iomem *reg;
    u32 id;
    u32 hw_id;
    u8 max_active_inputs;
    u8 max_active_outputs;
    u32 supported_inputs;
    u32 supported_outputs;
    const struct komeda_component_funcs *funcs;
};
```

Members

obj

treat component as private obj

pipeline

the komeda pipeline this component belongs to

name

component name

reg

component register base, which is initialized by chip and used by chip only

id

component id

hw_id

component hw id, which is initialized by chip and used by chip only

max_active_inputs**max_active_outputs:**

maximum number of inputs/outputs that can be active at the same time Note: the number isn't the bit number of **supported_inputs** or **supported_outputs**, but may be less than it, since component may not support enabling all **supported_inputs**/outputs at the same time.

max_active_outputs

maximum number of outputs

supported_inputs**supported_outputs:**

bitmask of BIT(component->id) for the supported inputs/outputs, describes the possibilities of how a component is linked into a pipeline.

supported_outputs

bitmask of supported output component ids

funcs

chip functions to access HW

Description

struct komeda_component describe the data flow capabilities for how to link a component into the display pipeline. all specified components are subclass of this structure.

struct **komeda_component_output**

Definition:

```
struct komeda_component_output {
    struct komeda_component *component;
    u8 output_port;
};
```

Members**component**

indicate which component the data comes from

output_port

the output port of the *komeda_component_output.component*

Description

a component has multiple outputs, if want to know where the data comes from, only know the component is not enough, we still need to know its output port

struct **komeda_component_state**

Definition:

```
struct komeda_component_state {
    struct drm_private_state obj;
    struct komeda_component *component;
    union {
        struct drm_crtc *crtc;
    };
};
```

```
    struct drm_plane *plane;
    struct drm_connector *wb_conn;
    void *binding_user;
};
u16 active_inputs;
u16 changed_active_inputs;
u16 affected_inputs;
struct komeda_component_output inputs[KOMEDA_COMPONENT_N_INPUTS];
};
```

Members

- obj**
tracking component_state by drm_atomic_state
- component**
backpointer to the component
- {unnamed_union}**
anonymous
- crtc**
backpointer for user crtc
- plane**
backpointer for user plane
- wb_conn**
backpointer for user wb_connector
- binding_user**
currently bound user, the user can be **crtc**, **plane** or **wb_conn**, which is valid decided by **component** and **inputs**
- Layer: its user always is plane.
 - compiz/improc/timing_ctrlr: the user is crtc.
 - wb_layer: wb_conn;
 - scaler: plane when input is layer, wb_conn if input is compiz.
- active_inputs**
active_inputs is bitmask of **inputs** index
- active_inputs = changed_active_inputs | unchanged_active_inputs
 - affected_inputs = old->active_inputs | new->active_inputs;
 - disabling_inputs = affected_inputs ^ active_inputs;
 - changed_inputs = disabling_inputs | changed_active_inputs;
- NOTE: changed_inputs doesn't include all active_input but only **changed_active_inputs**, and this bitmask can be used in chip level for dirty update.
- changed_active_inputs**
bitmask of the changed **active_inputs**

affected_inputs

bitmask for affected **inputs**

inputs

the specific inputs[i] only valid on BIT(i) has been set in **active_inputs**, if not the inputs[i] is undefined.

Description

component_state is the data flow configuration of the component, and it's the superclass of all specific component_state like **komeda_layer_state**, **komeda_scaler_state**

struct **komeda_pipeline**

Definition:

```
struct komeda_pipeline {
    struct drm_private_obj obj;
    struct komeda_dev *mdev;
    struct clk *pxlclk;
    int id;
    u32 avail_comps;
    u32 standalone_disabled_comps;
    int n_layers;
    struct komeda_layer *layers[KOMEDA_PIPELINE_MAX_LAYERS];
    int n_scalers;
    struct komeda_scaler *scalers[KOMEDA_PIPELINE_MAX_SCALERS];
    struct komeda_compiz *compiz;
    struct komeda_splitter *splitter;
    struct komeda_merger *merger;
    struct komeda_layer *wb_layer;
    struct komeda_improc *improc;
    struct komeda_timing_ctrlr *ctrlr;
    const struct komeda_pipeline_funcs *funcs;
    struct device_node *of_node;
    struct device_node *of_output_port;
    struct device_node *of_output_links[2];
    bool dual_link;
};
```

Members**obj**

link pipeline as private obj of drm_atomic_state

mdev

the parent komeda_dev

pxlclk

pixel clock

id

pipeline id

avail_comps

available components mask of pipeline

standalone_disabled_comps

When disable the pipeline, some components can not be disabled together with others, but need a sparated and standalone disable. The standalone_disabled_comps are the components which need to be disabled standalone, and this concept also introduce concept of two phase. phase 1: for disabling the common components. phase 2: for disabling the standalong_disabled_comps.

n_layers

the number of layer on **layers**

layers

the pipeline layers

n_scalers

the number of scaler on **scalers**

scalers

the pipeline scalers

compiz

compositor

splitter

for split the compiz output to two half data flows

merger

merger

wb_layer

writeback layer

improc

post image processor

ctrlr

timing controller

funcs

chip private pipeline functions

of_node

pipeline dt node

of_output_port

pipeline output port

of_output_links

output connector device nodes

dual_link

true if of_output_links[0] and [1] are both valid

Description

Represent a complete display pipeline and hold all functional components.

struct **komeda_pipeline_state**

Definition:


```
struct komeda_pipeline_state {
    struct drm_private_state obj;
    struct komeda_pipeline *pipe;
    struct drm_crtc *crtc;
    u32 active_comps;
};
```

Members

obj

tracking pipeline_state by drm_atomic_state

pipe

backpointer to the pipeline

crtc

currently bound crtc

active_comps

bitmask - BIT(component->id) of active components

NOTE

Unlike the pipeline, pipeline_state doesn't gather any component_state into it. It because all component will be managed by drm_atomic_state.

10.14.4 Resource discovery and initialization

Pipeline and component are used to describe how to handle the pixel data. We still need a `@struct komeda_dev` to describe the whole view of the device, and the control-abilites of device.

We have `&komeda_dev`, `&komeda_pipeline`, `&komeda_component`. Now fill devices with pipelines. Since komeda is not for D71 only but also intended for later products, of course we'd better share as much as possible between different products. To achieve this, split the komeda device into two layers: CORE and CHIP.

- CORE: for common features and capabilities handling.
- CHIP: for register programming and HW specific feature (limitation) handling.

CORE can access CHIP by three chip function structures:

- `struct komeda_dev_funcs`
- `struct komeda_pipeline_funcs`
- `struct komeda_component_funcs`

struct **komeda_dev_funcs**

Definition:

```
struct komeda_dev_funcs {
    void (*init_format_table)(struct komeda_dev *mdev);
    int (*enum_resources)(struct komeda_dev *mdev);
    void (*cleanup)(struct komeda_dev *mdev);
};
```

```
int (*connect_iommu)(struct komeda_dev *mdev);
int (*disconnect_iommu)(struct komeda_dev *mdev);
irqreturn_t (*irq_handler)(struct komeda_dev *mdev, struct komeda_events_
→*events);
int (*enable_irq)(struct komeda_dev *mdev);
int (*disable_irq)(struct komeda_dev *mdev);
void (*on_off_vblank)(struct komeda_dev *mdev, int master_pipe, bool on);
void (*dump_register)(struct komeda_dev *mdev, struct seq_file *seq);
int (*change_opmode)(struct komeda_dev *mdev, int new_mode);
void (*flush)(struct komeda_dev *mdev, int master_pipe, u32 active_pipes);
};
```

Members

init_format_table

initialize *komeda_dev->format_table*, this function should be called before the enum_resource

enum_resources

for CHIP to report or add pipeline and component resources to CORE

cleanup

call to chip to cleanup komeda_dev->chip data

connect_iommu

Optional, connect to external iommu

disconnect_iommu

Optional, disconnect to external iommu

irq_handler

for CORE to get the HW event from the CHIP when interrupt happened.

enable_irq

enable irq

disable_irq

disable irq

on_off_vblank

notify HW to on/off vblank

dump_register

Optional, dump registers to seq_file

change_opmode

Notify HW to switch to a new display operation mode.

flush

Notify the HW to flush or kickoff the update

Description

Supplied by chip level and returned by the chip entry function xxx_identify,

struct **komeda_dev**

Definition:

```

struct komeda_dev {
    struct device *dev;
    u32 __iomem *reg_base;
    struct komeda_chip_info chip;
    struct komeda_format_caps_table fmt_tbl;
    struct clk *aclk;
    int irq;
    struct mutex lock;
    u32 dpmode;
    int n_pipelines;
    struct komeda_pipeline *pipelines[KOMEDA_MAX_PIPELINES];
    const struct komeda_dev_funcs *funcs;
    void *chip_data;
    struct iommu_domain *iommu;
    struct dentry *debugfs_root;
    u16 err_verbosity;
#define KOMEDA_DEV_PRINT_ERR_EVENTS BIT(0);
#define KOMEDA_DEV_PRINT_WARN_EVENTS BIT(1);
#define KOMEDA_DEV_PRINT_INFO_EVENTS BIT(2);
#define KOMEDA_DEV_PRINT_DUMP_STATE_ON_EVENT BIT(8);
#define KOMEDA_DEV_PRINT_DISABLE_RATELIMIT BIT(12);
};

```

Members

dev

the base device structure

reg_base

the base address of komeda io space

chip

the basic chip information

fmt_tbl

initialized by *komeda_dev_funcs->init_format_table*

aclk

HW main engine clk

irq

irq number

lock

used to protect dpmode

dpmode

current display mode

n_pipelines

the number of pipe in **pipelines**

pipelines

the komeda pipelines

funcs

chip funcs to access to HW

chip_data

chip data will be added by `komeda_dev_funcs.enum_resources()` and destroyed by `komeda_dev_funcs.cleanup()`

iommu

iommu domain

debugfs_root

root directory of komeda debugfs

err_verbosity

bitmask for how much extra info to print on error

See KOMEDA_DEV_* macros for details. Low byte contains the debug level categories, the high byte contains extra debug options.

Description

Pipeline and component are used to describe how to handle the pixel data. `komeda_device` is for describing the whole view of the device, and the control-abilites of device.

10.14.5 Format handling

struct `komeda_format_caps`

Definition:

```
struct komeda_format_caps {
    u32 hw_id;
    u32 fourcc;
    u32 supported_layer_types;
    u32 supported_rots;
    u32 supported_afbc_layouts;
    u64 supported_afbc_features;
};
```

Members

hw_id

hw format id, hw specific value.

fourcc

drm fourcc format.

supported_layer_types

indicate which layer supports this format

supported_rots

allowed rotations for this format

supported_afbc_layouts

supported afbc layerout

supported_afbc_features

supported afbc features

Description

`komeda_format_caps` is for describing ARM display specific features and limitations for a specific format, and `format_caps` will be linked into `komeda_framebuffer` like a extension of [`drm_format_info`](#).

NOTE

one `fourcc` may has two different `format_caps` items for `fourcc` and `fourcc+modifier`

struct `komeda_format_caps_table`

`format_caps` mananger

Definition:

```
struct komeda_format_caps_table {
    u32 n_formats;
    const struct komeda_format_caps *format_caps;
    bool (*format_mod_supported)(const struct komeda_format_caps *caps, u32
→layer_type, u64 modifier, u32 rot);
};
```

Members

`n_formats`

the size of `format_caps` list.

`format_caps`

`format_caps` list.

`format_mod_supported`

Optional. Some HW may have special requirements or limitations which can not be described by `format_caps`, this func supply HW the ability to do the further HW specific check.

struct `komeda_fb`

Entending `drm_framebuffer` with `komeda` attribute

Definition:

```
struct komeda_fb {
    struct drm_framebuffer base;
    const struct komeda_format_caps *format_caps;
    bool is_va;
    u32 aligned_w;
    u32 aligned_h;
    u32 afbc_size;
    u32 offset_payload;
};
```

Members

`base`

[`drm_framebuffer`](#)

`format_caps`

extends `drm_format_info` for `komeda` specific information

is_va

if smmu is enabled, it will be true

aligned_w

aligned frame buffer width

aligned_h

aligned frame buffer height

afbc_size

minimum size of afbc

offset_payload

start of afbc body buffer

10.14.6 Attach komeda_dev to DRM-KMS

Komeda abstracts resources by pipeline/component, but DRM-KMS uses crtc/plane/connector. One KMS-obj cannot represent only one single component, since the requirements of a single KMS object cannot simply be achieved by a single component, usually that needs multiple components to fit the requirement. Like set mode, gamma, ctm for KMS all target on CRTC-obj, but komeda needs compiz, improc and timing_ctrlr to work together to fit these requirements. And a KMS-Plane may require multiple komeda resources: layer/scaler/compiz.

So, one KMS-Obj represents a sub-pipeline of komeda resources.

- Plane: *Layer(input) pipeline*
- Wb_connector: *Writeback(output) pipeline*
- Crtc: *Display output pipeline*

So, for komeda, we treat KMS crtc/plane/connector as users of pipeline and component, and at any one time a pipeline/component only can be used by one user. And pipeline/component will be treated as private object of DRM-KMS; the state will be managed by `drm_atomic_state` as well.

How to map plane to Layer(input) pipeline

Komeda has multiple Layer input pipelines, see: - *Single pipeline data flow* - *Dual pipeline with Slave enabled*

The easiest way is binding a plane to a fixed Layer pipeline, but consider the komeda capabilities:

- Layer Split, See *Layer(input) pipeline*

Layer_Split is quite complicated feature, which splits a big image into two parts and handles it by two layers and two scalers individually. But it imports an edge problem or effect in the middle of the image after the split. To avoid such a problem, it needs a complicated Split calculation and some special configurations to the layer and scaler. We'd better hide such HW related complexity to user mode.

- Slave pipeline, See *Dual pipeline with Slave enabled*

Since the compiz component doesn't output alpha value, the slave pipeline only can be used for bottom layers composition. The komeda driver wants to hide this limitation to the user. The way to do this is to pick a suitable Layer according to `plane_state->zpos`.

So for komeda, the KMS-plane doesn't represent a fixed komeda layer pipeline, but multiple Layers with same capabilities. Komeda will select one or more Layers to fit the requirement of one KMS-plane.

Make component/pipeline to be `drm_private_obj`

Add `drm_private_obj` to `komeda_component`, `komeda_pipeline`

```
struct komeda_component {
    struct drm_private_obj obj;
    ...
}

struct komeda_pipeline {
    struct drm_private_obj obj;
    ...
}
```

Tracking component_state/pipeline_state by `drm_atomic_state`

Add `drm_private_state` and user to `komeda_component_state`, `komeda_pipeline_state`

```
struct komeda_component_state {
    struct drm_private_state obj;
    void *binding_user;
    ...
}

struct komeda_pipeline_state {
    struct drm_private_state obj;
    struct drm_crtc *crtc;
    ...
}
```

komeda component validation

Komeda has multiple types of components, but the process of validation are similar, usually including the following steps:

```
int komeda_xxxx_validate(struct komeda_component_xxx xxx_comp,
    struct komeda_component_output *input_dflow,
    struct drm_plane/crtc/connector *user,
    struct drm_plane/crtc/connector_state, *user_state)
{
    setup 1: check if component is needed, like the scaler is optional_
```

↪ depending

on the user_state; **if** unneeded, just **return**, and the caller will put the data flow into next stage.

Setup 2: check user_state with component features and capabilities to see **if** requirements can be met; **if** not, **return** fail.

Setup 3: get component_state from drm_atomic_state, and try set to set user to component; fail **if** component has been assigned to another user already.

Setup 3: configure the component_state, like set its input component, convert user_state to component specific state.

Setup 4: adjust the input_dflow and prepare it **for** the next stage.

}

komeda_kms Abstraction

struct **komeda_plane**

komeda instance of drm_plane

Definition:

```
struct komeda_plane {
    struct drm_plane base;
    struct komeda_layer *layer;
};
```

Members

base

drm_plane

layer

represents available layer input pipelines for this plane.

NOTE: the layer is not for a specific Layer, but indicate a group of Layers with same capabilities.

struct **komeda_plane_state**

Definition:

```
struct komeda_plane_state {
    struct drm_plane_state base;
    struct list_head zlist_node;
    u8 layer_split : 1;
};
```

Members

base

drm_plane_state

zlist_node

zorder list node

layer_split

on/off layer_split

Description

The plane_state can be split into two data flow (left/right) and handled by two layers *komeda_plane.layer* and *komeda_plane.layer.right*

struct **komeda_wb_connector**

Definition:

```
struct komeda_wb_connector {
    struct drm_writeback_connector base;
    struct komeda_layer *wb_layer;
};
```

Members**base***drm_writeback_connector***wb_layer**

represents associated writeback pipeline of komeda

struct **komeda_crtc**

Definition:

```
struct komeda_crtc {
    struct drm_crtc base;
    struct komeda_pipeline *master;
    struct komeda_pipeline *slave;
    u32 slave_planes;
    struct komeda_wb_connector *wb_conn;
    struct completion *disable_done;
    struct drm_encoder encoder;
};
```

Members**base***drm_crtc***master**

only master has display output

slave

optional

Doesn't have its own display output, the handled data flow will merge into the master.

slave_planes

komeda slave planes mask

wb_conn

komeda write back connector

disable_done

this flip_done is for tracing the disable

encoder

encoder at the end of the pipeline

struct **komeda_crtc_state**

Definition:

```
struct komeda_crtc_state {
    struct drm_crtc_state base;
    u32 affected_pipes;
    u32 active_pipes;
    u64 clock_ratio;
    u32 max_slave_zorder;
};
```

Members

base

drm_crtc_state

affected_pipes

the affected pipelines in once display instance

active_pipes

the active pipelines in once display instance

clock_ratio

ratio of (aclk << 32)/pxlclk

max_slave_zorder

the maximum of slave zorder

komde_kms Functions

int **komeda_crtc_atomic_check**(struct *drm_crtc* *crtc, struct *drm_atomic_state* *state)

build display output data flow

Parameters

struct drm_crtc *crtc

DRM crtc

struct drm_atomic_state *state

the crtc state object

Description

crtc_atomic_check is the final check stage, so beside build a display data pipeline according to the crtc_state, but still needs to release or disable the unclaimed pipeline resources.

Return

Zero for success or -errno

int **komeda_plane_atomic_check**(struct *drm_plane* *plane, struct *drm_atomic_state* *state)
 build input data flow

Parameters

struct drm_plane *plane
 DRM plane

struct drm_atomic_state *state
 the plane state object

Return

Zero for success or -errno

10.14.7 Build komeda to be a Linux module driver

Now we have two level devices:

- **komeda_dev**: describes the real display hardware.
- **komeda_kms_dev**: attaches or connects **komeda_dev** to DRM-KMS.

All komeda operations are supplied or operated by **komeda_dev** or **komeda_kms_dev**, the module driver is only a simple wrapper to pass the Linux command (probe/remove/pm) into **komeda_dev** or **komeda_kms_dev**.

BACKLIGHT SUPPORT

The backlight core supports implementing backlight drivers.

A backlight driver registers a driver using `devm_backlight_device_register()`. The properties of the backlight driver such as type and `max_brightness` must be specified. When the core detect changes in for example brightness or power state the `update_status()` operation is called. The backlight driver shall implement this operation and use it to adjust backlight.

Several sysfs attributes are provided by the backlight core:

- | | |
|----------------------------------|--|
| - <code>brightness</code> | R/W, set the requested brightness level |
| - <code>actual_brightness</code> | R0, the brightness level used by the HW |
| - <code>max_brightness</code> | R0, the maximum brightness level supported |

See Documentation/ABI/stable/sysfs-class-backlight for the full list.

The backlight can be adjusted using the sysfs interface, and the backlight driver may also support adjusting backlight using a hot-key or some other platform or firmware specific way.

The driver must implement the `get_brightness()` operation if the HW do not support all the levels that can be specified in brightness, thus providing user-space access to the actual level via the `actual_brightness` attribute.

When the backlight changes this is reported to user-space using an uevent connected to the `actual_brightness` attribute. When brightness is set by platform specific means, for example a hot-key to adjust backlight, the driver must notify the backlight core that brightness has changed using `backlight_force_update()`.

The backlight driver core receives notifications from fbdev and if the event is `FB_EVENT_BLANK` and if the value of blank, from the `FBIOLANK` ioctl, results in a change in the backlight state the `update_status()` operation is called.

enum **backlight_update_reason**

what method was used to update backlight

Constants

BACKLIGHT_UPDATE_HOTKEY

The backlight was updated using a hot-key.

BACKLIGHT_UPDATE_SYSFS

The backlight was updated using sysfs.

Description

A driver indicates the method (reason) used for updating the backlight when calling *backlight_force_update()*.

enum **backlight_type**

the type of backlight control

Constants

BACKLIGHT_RAW

The backlight is controlled using hardware registers.

BACKLIGHT_PLATFORM

The backlight is controlled using a platform-specific interface.

BACKLIGHT_FIRMWARE

The backlight is controlled using a standard firmware interface.

BACKLIGHT_TYPE_MAX

Number of entries.

Description

The type of interface used to control the backlight.

enum **backlight_notification**

the type of notification

Constants

BACKLIGHT_REGISTERED

The backlight device is registered.

BACKLIGHT_UNREGISTERED

The backlight device is unregistered.

Description

The notifications that is used for notification sent to the receiver that registered notifications using *backlight_register_notifier()*.

struct **backlight_ops**

backlight operations

Definition:

```
struct backlight_ops {
    unsigned int options;
#define BL_CORE_SUSPENDRESUME    (1 << 0);
    int (*update_status)(struct backlight_device *);
    int (*get_brightness)(struct backlight_device *);
    int (*check_fb)(struct backlight_device *bd, struct fb_info *info);
};
```

Members

options

Configure how operations are called from the core.

The `options` parameter is used to adjust the behaviour of the core. Set `BL_CORE_SUSPENDRESUME` to get the `update_status()` operation called upon suspend and resume.

update_status

Operation called when properties have changed.

Notify the backlight driver some property has changed. The `update_status` operation is protected by the `update_lock`.

The backlight driver is expected to use `backlight_is_blank()` to check if the display is blanked and set brightness accordingly. `update_status()` is called when any of the properties has changed.

RETURNS:

0 on success, negative error code if any failure occurred.

get_brightness

Return the current backlight brightness.

The driver may implement this as a readback from the HW. This operation is optional and if not present then the current brightness property value is used.

RETURNS:

A brightness value which is 0 or a positive number. On failure a negative error code is returned.

check_fb

Check the framebuffer device.

Check if given framebuffer device is the one bound to this backlight. This operation is optional and if not implemented it is assumed that the `fbdev` is always the one bound to the backlight.

RETURNS:

If `info` is `NULL` or the `info` matches the `fbdev` bound to the backlight return true. If `info` does not match the `fbdev` bound to the backlight return false.

Description

The backlight operations are specified when the backlight device is registered.

struct **backlight_properties**

backlight properties

Definition:

```
struct backlight_properties {
    int brightness;
    int max_brightness;
    int power;
    int fb_blank;
    enum backlight_type type;
    unsigned int state;
#define BL_CORE_SUSPENDED      (1 << 0)
#define BL_CORE_FBBLANK       (1 << 1)
```

```
enum backlight_scale scale;
};
```

Members

brightness

The current brightness requested by the user.

The backlight core makes sure the range is (0 to max_brightness) when the brightness is set via the sysfs attribute: /sys/class/backlight/<backlight>/brightness.

This value can be set in the backlight_properties passed to [devm_backlight_device_register\(\)](#) to set a default brightness value.

max_brightness

The maximum brightness value.

This value must be set in the backlight_properties passed to [devm_backlight_device_register\(\)](#) and shall not be modified by the driver after registration.

power

The current power mode.

User space can configure the power mode using the sysfs attribute: /sys/class/backlight/<backlight>/bl_power. When the power property is updated [update_status\(\)](#) is called.

The possible values are: (0: full on, 1 to 3: power saving modes; 4: full off), see FB_BLANK_XXX.

When the backlight device is enabled **power** is set to FB_BLANK_UNBLANK. When the backlight device is disabled **power** is set to FB_BLANK_POWERDOWN.

fb_blank

The power state from the FBIOLANK ioctl.

When the FBIOLANK ioctl is called **fb_blank** is set to the blank parameter and the [update_status\(\)](#) operation is called.

When the backlight device is enabled **fb_blank** is set to FB_BLANK_UNBLANK. When the backlight device is disabled **fb_blank** is set to FB_BLANK_POWERDOWN.

Backlight drivers should avoid using this property. It has been replaced by state & BL_CORE_FBLANK (although most drivers should use [backlight_is_blank\(\)](#) as the preferred means to get the blank state).

fb_blank is deprecated and will be removed.

type

The type of backlight supported.

The backlight type allows userspace to make appropriate policy decisions based on the backlight type.

This value must be set in the backlight_properties passed to [devm_backlight_device_register\(\)](#).

state

The state of the backlight core.

The state is a bitmask. `BL_CORE_FBBLANK` is set when the display is expected to be blank. `BL_CORE_SUSPENDED` is set when the driver is suspended.

backlight drivers are expected to use `backlight_is_blank()` in their `update_status()` operation rather than reading the state property.

The state is maintained by the core and drivers may not modify it.

scale

The type of the brightness scale.

Description

This structure defines all the properties of a backlight.

struct **backlight_device**

backlight device data

Definition:

```
struct backlight_device {
    struct backlight_properties props;
    struct mutex update_lock;
    struct mutex ops_lock;
    const struct backlight_ops *ops;
    struct notifier_block fb_notif;
    struct list_head entry;
    struct device dev;
    bool fb_bl_on[FB_MAX];
    int use_count;
};
```

Members

props

Backlight properties

update_lock

The lock used when calling the `update_status()` operation.

`update_lock` is an internal backlight lock that serialise access to the `update_status()` operation. The backlight core holds the `update_lock` when calling the `update_status()` operation. The `update_lock` shall not be used by backlight drivers.

ops_lock

The lock used around everything related to `backlight_ops`.

`ops_lock` is an internal backlight lock that protects the `ops` pointer and is used around all accesses to `ops` and when the operations are invoked. The `ops_lock` shall not be used by backlight drivers.

ops

Pointer to the backlight operations.

If `ops` is `NULL`, the driver that registered this device has been unloaded, and if `class_get_devdata()` points to something in the body of that driver, it is also invalid.

fb_notif

The framebuffer notifier block

entry

List entry of all registered backlight devices

dev

Parent device.

fb_bl_on

The state of individual fbdev's.

Multiple fbdev's may share one backlight device. The fb_bl_on records the state of the individual fbdev.

use_count

The number of uses of fb_bl_on.

Description

This structure holds all data required by a backlight device.

int **backlight_update_status**(struct *backlight_device* *bd)

force an update of the backlight device status

Parameters

struct **backlight_device** *bd

the backlight device

int **backlight_enable**(struct *backlight_device* *bd)

Enable backlight

Parameters

struct **backlight_device** *bd

the backlight device to enable

int **backlight_disable**(struct *backlight_device* *bd)

Disable backlight

Parameters

struct **backlight_device** *bd

the backlight device to disable

bool **backlight_is_blank**(const struct *backlight_device* *bd)

Return true if display is expected to be blank

Parameters

const struct **backlight_device** *bd

the backlight device

Description

Display is expected to be blank if any of these is true:

- 1) if power in not UNBLANK
- 2) if fb_blank is not UNBLANK
- 3) if state indicate BLANK or SUSPENDED

Returns true if display is expected to be blank, false otherwise.

int **backlight_get_brightness**(const struct *backlight_device* *bd)

Returns the current brightness value

Parameters

const struct **backlight_device** *bd

the backlight device

Description

Returns the current brightness value, taking in consideration the current state. If *backlight_is_blank()* returns true then return 0 as brightness otherwise return the current brightness property value.

Backlight drivers are expected to use this function in their update_status() operation to get the brightness value.

void ***bl_get_data**(struct *backlight_device* *bl_dev)

access devdata

Parameters

struct **backlight_device** *bl_dev

pointer to backlight device

Description

When a backlight device is registered the driver has the possibility to supply a void * devdata. *bl_get_data()* return a pointer to the devdata.

pointer to devdata stored while registering the backlight device.

Return

void **backlight_force_update**(struct *backlight_device* *bd, enum *backlight_update_reason* reason)

tell the backlight subsystem that hardware state has changed

Parameters

struct **backlight_device** *bd

the backlight device to update

enum **backlight_update_reason** reason

reason for update

Description

Updates the internal state of the backlight in response to a hardware event, and generates an uevent to notify userspace. A backlight driver shall call *backlight_force_update()* when the backlight is changed using, for example, a hot-key. The updated brightness is read using get_brightness() and the brightness value is reported using an uevent.

struct *backlight_device* ***backlight_device_get_by_name**(const char *name)

Get backlight device by name

Parameters

const char *name

Device name

Description

This function looks up a backlight device by its name. It obtains a reference on the backlight device and it is the caller's responsibility to drop the reference by calling `put_device()`.

Return

A pointer to the backlight device if found, otherwise NULL.

int **backlight_register_notifier**(struct notifier_block *nb)
get notified of backlight (un)registration

Parameters

struct notifier_block *nb
notifier block with the notifier to call on backlight (un)registration

Description

Register a notifier to get notified when backlight devices get registered or unregistered.

0 on success, otherwise a negative error code

Return

int **backlight_unregister_notifier**(struct notifier_block *nb)
unregister a backlight notifier

Parameters

struct notifier_block *nb
notifier block to unregister

Description

Register a notifier to get notified when backlight devices get registered or unregistered.

0 on success, otherwise a negative error code

Return

struct *backlight_device* *devm_backlight_device_register(struct device *dev, const char *name, struct device *parent, void *devdata, const struct *backlight_ops* *ops, const struct *backlight_properties* *props)
register a new backlight device

Parameters

struct device *dev
the device to register

const char *name
the name of the device

struct device *parent
a pointer to the parent device (often the same as **dev**)

void *devdata
an optional pointer to be stored for private driver use

const struct backlight_ops *ops
the backlight operations structure

const struct backlight_properties *props
the backlight properties

Description

Creates and registers new backlight device. When a backlight device is registered the configuration must be specified in the **props** parameter. See description of [backlight_properties](#).

struct backlight on success, or an ERR_PTR on error

Return

void **devm_backlight_device_unregister**(struct device *dev, struct [backlight_device](#) *bd)
unregister backlight device

Parameters

struct device *dev
the device to unregister

struct backlight_device *bd
the backlight device to unregister

Description

Deallocates a backlight allocated with [devm_backlight_device_register\(\)](#). Normally this function will not need to be called and the resource management code will ensure that the resources are freed.

struct [backlight_device](#) ***of_find_backlight_by_node**(struct device_node *node)
find backlight device by device-tree node

Parameters

struct device_node *node
device-tree node of the backlight device

Description

Returns a pointer to the backlight device corresponding to the given DT node or NULL if no such backlight device exists or if the device hasn't been probed yet.

This function obtains a reference on the backlight device and it is the caller's responsibility to drop the reference by calling `put_device()` on the backlight device's `.dev` field.

struct [backlight_device](#) ***devm_of_find_backlight**(struct device *dev)
find backlight for a device

Parameters

struct device *dev
the device

Description

This function looks for a property named 'backlight' on the DT node connected to **dev** and looks up the backlight device. The lookup is device managed so the reference to the backlight device is automatically dropped on driver detach.

A pointer to the backlight device if found. Error pointer -EPROBE_DEFER if the DT property is set, but no backlight device is found. NULL if there's no backlight property.

Return

VGA SWITCHEROO

`vga_switcheroo` is the Linux subsystem for laptop hybrid graphics. These come in two flavors:

- **muxed:** Dual GPUs with a multiplexer chip to switch outputs between GPUs.
- **muxless:** Dual GPUs but only one of them is connected to outputs. The other one is merely used to offload rendering, its results are copied over PCIe into the framebuffer. On Linux this is supported with DRI PRIME.

Hybrid graphics started to appear in the late Naughties and were initially all muxed. Newer laptops moved to a muxless architecture for cost reasons. A notable exception is the MacBook Pro which continues to use a mux. Muxes come with varying capabilities: Some switch only the panel, others can also switch external displays. Some switch all display pins at once while others can switch just the DDC lines. (To allow EDID probing for the inactive GPU.) Also, muxes are often used to cut power to the discrete GPU while it is not used.

DRM drivers register GPUs with `vga_switcheroo`, these are henceforth called clients. The mux is called the handler. Muxless machines also register a handler to control the power state of the discrete GPU, its `->switchto` callback is a no-op for obvious reasons. The discrete GPU is often equipped with an HDA controller for the HDMI/DP audio signal, this will also register as a client so that `vga_switcheroo` can take care of the correct suspend/resume order when changing the discrete GPU's power state. In total there can thus be up to three clients: Two `vga` clients (GPUs) and one audio client (on the discrete GPU). The code is mostly prepared to support machines with more than two GPUs should they become available.

The GPU to which the outputs are currently switched is called the active client in `vga_switcheroo` parlance. The GPU not in use is the inactive client. When the inactive client's DRM driver is loaded, it will be unable to probe the panel's EDID and hence depends on VBIOS to provide its display modes. If the VBIOS modes are bogus or if there is no VBIOS at all (which is common on the MacBook Pro), a client may alternatively request that the DDC lines are temporarily switched to it, provided that the handler supports this. Switching only the DDC lines and not the entire output avoids unnecessary flickering.

12.1 Modes of Use

12.1.1 Manual switching and manual power control

In this mode of use, the file `/sys/kernel/debug/vgaswitcheroo/switch` can be read to retrieve the current `vga_switcheroo` state and commands can be written to it to change the state. The file appears as soon as two GPU drivers and one handler have registered with `vga_switcheroo`. The following commands are understood:

- OFF: Power off the device not in use.
- ON: Power on the device not in use.
- IGD: Switch to the integrated graphics device. Power on the integrated GPU if necessary, power off the discrete GPU. Prerequisite is that no user space processes (e.g. Xorg, alsactl) have opened device files of the GPUs or the audio client. If the switch fails, the user may invoke `lsof(8)` or `fuser(1)` on `/dev/dri/` and `/dev/snd/controlC1` to identify processes blocking the switch.
- DIS: Switch to the discrete graphics device.
- DIGD: Delayed switch to the integrated graphics device. This will perform the switch once the last user space process has closed the device files of the GPUs and the audio client.
- DDIS: Delayed switch to the discrete graphics device.
- MIGD: Mux-only switch to the integrated graphics device. Does not remap console or change the power state of either gpu. If the integrated GPU is currently off, the screen will turn black. If it is on, the screen will show whatever happens to be in VRAM. Either way, the user has to blindly enter the command to switch back.
- MDIS: Mux-only switch to the discrete graphics device.

For GPUs whose power state is controlled by the driver's runtime pm, the ON and OFF commands are a no-op (see next section).

For muxless machines, the IGD/DIS, DIGD/DDIS and MIGD/MDIS commands should not be used.

12.1.2 Driver power control

In this mode of use, the discrete GPU automatically powers up and down at the discretion of the driver's runtime pm. On muxed machines, the user may still influence the muxer state by way of the debugfs interface, however the ON and OFF commands become a no-op for the discrete GPU.

This mode is the default on Nvidia HybridPower/Optimus and ATI PowerXpress. Specifying `nouveau.runpm=0`, `radeon.runpm=0` or `amdgpu.runpm=0` on the kernel command line disables it.

After the GPU has been suspended, the handler needs to be called to cut power to the GPU. Likewise it needs to reinstate power before the GPU can resume. This is achieved by `vga_switcheroo_init_domain_pm_ops()`, which augments the GPU's suspend/resume functions by the requisite calls to the handler.

When the audio device resumes, the GPU needs to be woken. This is achieved by a PCI quirk which calls `device_link_add()` to declare a dependency on the GPU. That way, the GPU is kept awake whenever and as long as the audio device is in use.

On muxed machines, if the mux is initially switched to the discrete GPU, the user ends up with a black screen when the GPU powers down after boot. As a workaround, the mux is forced to the integrated GPU on runtime suspend, cf. https://bugs.freedesktop.org/show_bug.cgi?id=75917

12.2 API

12.2.1 Public functions

int **vga_switcheroo_register_handler**(const struct *vga_switcheroo_handler* *handler, enum *vga_switcheroo_handler_flags_t* handler_flags)

register handler

Parameters

const struct **vga_switcheroo_handler** *handler
handler callbacks

enum **vga_switcheroo_handler_flags_t** handler_flags
handler flags

Description

Register handler. Enable `vga_switcheroo` if two vga clients have already registered.

Return

0 on success, -EINVAL if a handler was already registered.

void **vga_switcheroo_unregister_handler**(void)
unregister handler

Parameters

void
no arguments

Description

Unregister handler. Disable `vga_switcheroo`.

enum *vga_switcheroo_handler_flags_t* **vga_switcheroo_handler_flags**(void)
obtain handler flags

Parameters

void
no arguments

Description

Helper for clients to obtain the handler flags bitmask.

Return

Handler flags. A value of 0 means that no handler is registered or that the handler has no special capabilities.

```
int vga_switcheroo_register_client(struct pci_dev *pdev, const struct
                                vga_switcheroo_client_ops *ops, bool
                                driver_power_control)
```

register vga client

Parameters

struct pci_dev *pdev
client pci device

const struct vga_switcheroo_client_ops *ops
client callbacks

bool driver_power_control
whether power state is controlled by the driver's runtime pm

Description

Register vga client (GPU). Enable vga_switcheroo if another GPU and a handler have already registered. The power state of the client is assumed to be ON. Beforehand, [vga_switcheroo_client_probe_defer\(\)](#) shall be called to ensure that all prerequisites are met.

Return

0 on success, -ENOMEM on memory allocation error.

```
int vga_switcheroo_register_audio_client(struct pci_dev *pdev, const struct
                                        vga_switcheroo_client_ops *ops, struct pci_dev
                                        *vga_dev)
```

register audio client

Parameters

struct pci_dev *pdev
client pci device

const struct vga_switcheroo_client_ops *ops
client callbacks

struct pci_dev *vga_dev
pci device which is bound to current audio client

Description

Register audio client (audio device on a GPU). The client is assumed to use runtime PM. Beforehand, [vga_switcheroo_client_probe_defer\(\)](#) shall be called to ensure that all prerequisites are met.

Return

0 on success, -ENOMEM on memory allocation error, -EINVAL on getting client id error.

bool vga_switcheroo_client_probe_defer(struct pci_dev *pdev)
whether to defer probing a given client

Parameters

struct pci_dev *pdev
client pci device

Description

Determine whether any prerequisites are not fulfilled to probe a given client. Drivers shall invoke this early on in their ->probe callback and return -EPROBE_DEFER if it evaluates to true. Thou shalt not register the client ere thou hast called this.

Return

true if probing should be deferred, otherwise false.

enum *vga_switcheroo_state* **vga_switcheroo_get_client_state**(struct pci_dev *pdev)
obtain power state of a given client

Parameters

struct pci_dev *pdev
client pci device

Description

Obtain power state of a given client as seen from vga_switcheroo. The function is only called from hda_intel.c.

Return

Power state.

void **vga_switcheroo_unregister_client**(struct pci_dev *pdev)
unregister client

Parameters

struct pci_dev *pdev
client pci device

Description

Unregister client. Disable vga_switcheroo if this is a vga client (GPU).

void **vga_switcheroo_client_fb_set**(struct pci_dev *pdev, struct fb_info *info)
set framebuffer of a given client

Parameters

struct pci_dev *pdev
client pci device

struct fb_info *info
framebuffer

Description

Set framebuffer of a given client. The console will be remapped to this on switching.

int **vga_switcheroo_lock_ddc**(struct pci_dev *pdev)
temporarily switch DDC lines to a given client

Parameters

struct pci_dev *pdev
client pci device

Description

Temporarily switch DDC lines to the client identified by **pdev** (but leave the outputs otherwise switched to where they are). This allows the inactive client to probe EDID. The DDC lines must afterwards be switched back by calling [vga_switcheroo_unlock_ddc\(\)](#), even if this function returns an error.

Return

Previous DDC owner on success or a negative int on error. Specifically, -ENODEV if no handler has registered or if the handler does not support switching the DDC lines. Also, a negative value returned by the handler is propagated back to the caller. The return value has merely an informational purpose for any caller which might be interested in it. It is acceptable to ignore the return value and simply rely on the result of the subsequent EDID probe, which will be NULL if DDC switching failed.

int **vga_switcheroo_unlock_ddc**(struct pci_dev *pdev)
switch DDC lines back to previous owner

Parameters

struct pci_dev *pdev
client pci device

Description

Switch DDC lines back to the previous owner after calling [vga_switcheroo_lock_ddc\(\)](#). This must be called even if [vga_switcheroo_lock_ddc\(\)](#) returned an error.

Return

Previous DDC owner on success (i.e. the client identifier of **pdev**) or a negative int on error. Specifically, -ENODEV if no handler has registered or if the handler does not support switching the DDC lines. Also, a negative value returned by the handler is propagated back to the caller. Finally, invoking this function without calling [vga_switcheroo_lock_ddc\(\)](#) first is not allowed and will result in -EINVAL.

int **vga_switcheroo_process_delayed_switch**(void)
helper for delayed switching

Parameters

void
no arguments

Description

Process a delayed switch if one is pending. DRM drivers should call this from their ->lastclose callback.

Return

0 on success. -EINVAL if no delayed switch is pending, if the client has unregistered in the meantime or if there are other clients blocking the switch. If the actual switch fails, an error is reported and 0 is returned.

```
int vga_switcheroo_init_domain_pm_ops(struct device *dev, struct dev_pm_domain
                                     *domain)
```

helper for driver power control

Parameters

struct device *dev

vga client device

struct dev_pm_domain *domain

power domain

Description

Helper for GPUs whose power state is controlled by the driver's runtime pm. After the GPU has been suspended, the handler needs to be called to cut power to the GPU. Likewise it needs to reinstate power before the GPU can resume. To this end, this helper augments the suspend/resume functions by the requisite calls to the handler. It needs only be called on platforms where the power switch is separate to the device being powered down.

12.2.2 Public structures

struct **vga_switcheroo_handler**

handler callbacks

Definition:

```
struct vga_switcheroo_handler {
    int (*init)(void);
    int (*switchto)(enum vga_switcheroo_client_id id);
    int (*switch_ddc)(enum vga_switcheroo_client_id id);
    int (*power_state)(enum vga_switcheroo_client_id id, enum vga_switcheroo_
↪state state);
    enum vga_switcheroo_client_id (*get_client_id)(struct pci_dev *pdev);
};
```

Members

init

initialize handler. Optional. This gets called when vga_switcheroo is enabled, i.e. when two vga clients have registered. It allows the handler to perform some delayed initialization that depends on the existence of the vga clients. Currently only the radeon and amdgpu drivers use this. The return value is ignored

switchto

switch outputs to given client. Mandatory. For muxless machines this should be a no-op. Returning 0 denotes success, anything else failure (in which case the switch is aborted)

switch_ddc

switch DDC lines to given client. Optional. Should return the previous DDC owner on success or a negative int on failure

power_state

cut or reinstate power of given client. Optional. The return value is ignored

get_client_id

determine if given pci device is integrated or discrete GPU. Mandatory

Description

Handler callbacks. The multiplexer itself. The **switchto** and **get_client_id** methods are mandatory, all others may be set to NULL.

struct **vga_switcheroo_client_ops**

client callbacks

Definition:

```
struct vga_switcheroo_client_ops {
    void (*set_gpu_state)(struct pci_dev *dev, enum vga_switcheroo_state);
    void (*reprobe)(struct pci_dev *dev);
    bool (*can_switch)(struct pci_dev *dev);
    void (*gpu_bound)(struct pci_dev *dev, enum vga_switcheroo_client_id);
};
```

Members

set_gpu_state

do the equivalent of suspend/resume for the card. Mandatory. This should not cut power to the discrete GPU, which is the job of the handler

reprobe

poll outputs. Optional. This gets called after waking the GPU and switching the outputs to it

can_switch

check if the device is in a position to switch now. Mandatory. The client should return false if a user space process has one of its device files open

gpu_bound

notify the client id to audio client when the GPU is bound.

Description

Client callbacks. A client can be either a GPU or an audio device on a GPU. The **set_gpu_state** and **can_switch** methods are mandatory, **reprobe** may be set to NULL. For audio clients, the **reprobe** member is bogus. OTOH, **gpu_bound** is only for audio clients, and not used for GPU clients.

12.2.3 Public constants

enum **vga_switcheroo_handler_flags_t**

handler flags bitmask

Constants

VGA_SWITCHEROO_CAN_SWITCH_DDC

whether the handler is able to switch the DDC lines separately. This signals to clients that they should call [*drm_get_edid_switcheroo\(\)*](#) to probe the EDID

VGA_SWITCHEROO_NEEDS_EDP_CONFIG

whether the handler is unable to switch the AUX channel separately. This signals to clients

that the active GPU needs to train the link and communicate the link parameters to the inactive GPU (mediated by `vga_switcheroo`). The inactive GPU may then skip the AUX handshake and set up its output with these pre-calibrated values (DisplayPort specification v1.1a, section 2.5.3.3)

Description

Handler flags bitmask. Used by handlers to declare their capabilities upon registering with `vga_switcheroo`.

enum **vga_switcheroo_client_id**
client identifier

Constants

VGA_SWITCHEROO_UNKNOWN_ID
initial identifier assigned to vga clients. Determining the id requires the handler, so GPUs are given their true id in a delayed fashion in `vga_switcheroo_enable()`

VGA_SWITCHEROO_IGD
integrated graphics device

VGA_SWITCHEROO_DIS
discrete graphics device

VGA_SWITCHEROO_MAX_CLIENTS
currently no more than two GPUs are supported

Description

Client identifier. Audio clients use the same identifier & 0x100.

enum **vga_switcheroo_state**
client power state

Constants

VGA_SWITCHEROO_OFF
off

VGA_SWITCHEROO_ON
on

VGA_SWITCHEROO_NOT_FOUND
client has not registered with `vga_switcheroo`. Only used in `vga_switcheroo_get_client_state()` which in turn is only called from `hda_intel.c`

Description

Client power state.

12.2.4 Private structures

struct **vgasr_priv**

vga_switcheroo private data

Definition:

```
struct vgasr_priv {
    bool active;
    bool delayed_switch_active;
    enum vga_switcheroo_client_id delayed_client_id;
    struct dentry *debugfs_root;
    int registered_clients;
    struct list_head clients;
    const struct vga_switcheroo_handler *handler;
    enum vga_switcheroo_handler_flags_t handler_flags;
    struct mutex mux_hw_lock;
    int old_ddc_owner;
};
```

Members

active

whether vga_switcheroo is enabled. Prerequisite is the registration of two GPUs and a handler

delayed_switch_active

whether a delayed switch is pending

delayed_client_id

client to which a delayed switch is pending

debugfs_root

directory for vga_switcheroo debugfs interface

registered_clients

number of registered GPUs (counting only vga clients, not audio clients)

clients

list of registered clients

handler

registered handler

handler_flags

flags of registered handler

mux_hw_lock

protects mux state (in particular while DDC lines are temporarily switched)

old_ddc_owner

client to which DDC lines will be switched back on unlock

Description

vga_switcheroo private data. Currently only one vga_switcheroo instance per system is supported.

struct **vga_switcheroo_client**

registered client

Definition:

```
struct vga_switcheroo_client {
    struct pci_dev *pdev;
    struct fb_info *fb_info;
    enum vga_switcheroo_state pwr_state;
    const struct vga_switcheroo_client_ops *ops;
    enum vga_switcheroo_client_id id;
    bool active;
    bool driver_power_control;
    struct list_head list;
    struct pci_dev *vga_dev;
};
```

Members

pdev

client pci device

fb_info

framebuffer to which console is remapped on switching

pwr_state

current power state if manual power control is used. For driver power control, call `vga_switcheroo_pwr_state()`.

ops

client callbacks

id

client identifier. Determining the id requires the handler, so gpus are initially assigned `VGA_SWITCHEROO_UNKNOWN_ID` and later given their true id in `vga_switcheroo_enable()`

active

whether the outputs are currently switched to this client

driver_power_control

whether power state is controlled by the driver's runtime pm. If true, writing ON and OFF to the `vga_switcheroo debugfs` interface is a no-op so as not to interfere with runtime pm

list

client list

vga_dev

pci device, indicate which GPU is bound to current audio client

Description

Registered client. A client can be either a GPU or an audio device on a GPU. For audio clients, the **fb_info** and **active** members are bogus. For GPU clients, the **vga_dev** is bogus.

12.3 Handlers

12.3.1 apple-gmux Handler

gmux is a microcontroller built into the MacBook Pro to support dual GPUs: A [Lattice XP2](#) on pre-retinas, a [Renesas R4F2113](#) on pre-T2 retinas.

On T2 Macbooks, the gmux is part of the T2 Coprocessor's SMC. The SMC has an I2C connection to a *NXP PCAL6524* GPIO expander, which enables/disables the voltage regulators of the discrete GPU, drives the display panel power, and has a GPIO to switch the eDP mux. The Intel CPU can interact with gmux through MMIO, similar to how the main SMC interface is controlled.

(The MacPro6,1 2013 also has a gmux, however it is unclear why since it has dual GPUs but no built-in display.)

gmux is connected to the LPC bus of the southbridge. Its I/O ports are accessed differently depending on the microcontroller: Driver functions to access a pre-retina gmux are infixed `_pio_`, those for a pre-T2 retina gmux are infixed `_index_`, and those on T2 Macs are infixed with `_mmio_`.

gmux is also connected to a GPIO pin of the southbridge and thereby is able to trigger an ACPI GPE. ACPI name GMGP holds this GPIO pin's number. On the MBP5 2008/09 it's GPIO pin 22 of the Nvidia MCP79, on following generations it's GPIO pin 6 of the Intel PCH, on MMIO gmux's it's pin 21.

The GPE merely signals that an interrupt occurred, the actual type of event is identified by reading a gmux register.

In addition to the GMGP name, gmux's ACPI device also has two methods GMSP and GMLV. GMLV likely means "GMUX Level", and reads the value of the GPIO, while GMSP likely means "GMUX Set Polarity", and seems to write to the GPIO's value. On newer Macbooks (This was introduced with or sometime before the MacBookPro14,3), the ACPI GPE method differentiates between the OS type: On Darwin, only a notification is signaled, whereas on other OSes, the GPIO's value is read and then inverted.

Because Linux masquerades as Darwin, it ends up in the notification-only code path. On MMIO gmux's, this seems to lead to us being unable to clear interrupts, unless we call GMSP(0). Without this, there is a flood of status=0 interrupts that can't be cleared. This issue seems to be unique to MMIO gmux's.

Graphics mux

On pre-retinas, the LVDS outputs of both GPUs feed into gmux which muxes either of them to the panel. One of the tricks gmux has up its sleeve is to lengthen the blanking interval of its output during a switch to synchronize it with the GPU switched to. This allows for a flicker-free switch that is imperceptible by the user ([US 8,687,007 B2](#)).

On retinas, muxing is no longer done by gmux itself, but by a separate chip which is controlled by gmux. The chip is triple sourced, it is either an [NXP CBTL06142](#), [TI HD3SS212](#) or [Pericom PI3VDP12412](#). The panel is driven with eDP instead of LVDS since the pixel clock required for retina resolution exceeds LVDS' limits.

Pre-retinas are able to switch the panel's DDC pins separately. This is handled by a [TI SN74LV4066A](#) which is controlled by gmux. The inactive GPU can thus probe the panel's EDID without switching over the entire panel. Retinas lack this functionality as the chips used for eDP muxing are incapable of switching the AUX channel separately (see the linked data sheets, Pericom would be capable but this is unused). However the retina panel has the NO_AUX_HANDSHAKE_LINK_TRAINING bit set in its DPCD, allowing the inactive GPU to skip the AUX handshake and set up the output with link parameters pre-calibrated by the active GPU.

The external DP port is only fully switchable on the first two unibody MacBook Pro generations, MBP5 2008/09 and MBP6 2010. This is done by an [NXP CBTL06141](#) which is controlled by gmux. It's the predecessor of the eDP mux on retinas, the difference being support for 2.7 versus 5.4 Gbit/s.

The following MacBook Pro generations replaced the external DP port with a combined DP/Thunderbolt port and lost the ability to switch it between GPUs, connecting it either to the discrete GPU or the Thunderbolt controller. Oddly enough, while the full port is no longer switchable, AUX and HPD are still switchable by way of an [NXP CBTL03062](#) (on pre-retinas MBP8 2011 and MBP9 2012) or two [TI TS3DS10224](#) (on pre-t2 retinas) under the control of gmux. Since the integrated GPU is missing the main link, external displays appear to it as phantoms which fail to link-train.

gmux receives the HPD signal of all display connectors and sends an interrupt on hotplug. On generations which cannot switch external ports, the discrete GPU can then be woken to drive the newly connected display. The ability to switch AUX on these generations could be used to improve reliability of hotplug detection by having the integrated GPU poll the ports while the discrete GPU is asleep, but currently we do not make use of this feature.

Our switching policy for the external port is that on those generations which are able to switch it fully, the port is switched together with the panel when IGD / DIS commands are issued to vga_switcheroo. It is thus possible to drive e.g. a beamer on battery power with the integrated GPU. The user may manually switch to the discrete GPU if more performance is needed.

On all newer generations, the external port can only be driven by the discrete GPU. If a display is plugged in while the panel is switched to the integrated GPU, *both* GPUs will be in use for maximum performance. To decrease power consumption, the user may manually switch to the discrete GPU, thereby suspending the integrated GPU.

gmux' initial switch state on bootup is user configurable via the EFI variable gpu-power-prefs-fa4ce28d-b62f-4c99-9cc3-6815686e30f9 (5th byte, 1 = IGD, 0 = DIS). Based on this setting, the EFI firmware tells gmux to switch the panel and the external DP connector and allocates a framebuffer for the selected GPU.

Power control

gmux is able to cut power to the discrete GPU. It automatically takes care of the correct sequence to tear down and bring up the power rails for core voltage, VRAM and PCIe.

Backlight control

On single GPU MacBooks, the PWM signal for the backlight is generated by the GPU. On dual GPU MacBook Pros by contrast, either GPU may be suspended to conserve energy. Hence the PWM signal needs to be generated by a separate backlight driver which is controlled by gmux. The earliest generation MBP5 2008/09 uses a [TI LP8543](#) backlight driver. Newer models use a [TI LP8545](#) or a TI LP8548.

Public functions

bool **apple_gmux_detect**(struct [pnp_dev](#) *pnp_dev, enum apple_gmux_type *type_ret)
detect if gmux is built into the machine

Parameters

struct pnp_dev *pnp_dev
Device to probe or NULL to use the first matching device

enum apple_gmux_type *type_ret
Returns (by reference) the apple_gmux_type of the device

Description

Detect if a supported gmux device is present by actually probing it. This avoids the false positives returned on some models by [apple_gmux_present\(\)](#).

Return

true if a supported gmux ACPI device is detected and the kernel was configured with CONFIG_APPLE_GMUX, false otherwise.

bool **apple_gmux_present**(void)
check if gmux ACPI device is present

Parameters

void
no arguments

Description

Drivers may use this to activate quirks specific to dual GPU MacBook Pros and Mac Pros, e.g. for deferred probing, runtime pm and backlight.

Return

true if gmux ACPI device is present and the kernel was configured with CONFIG_APPLE_GMUX, false otherwise.

VGA ARBITER

Graphic devices are accessed through ranges in I/O or memory space. While most modern devices allow relocation of such ranges, some “Legacy” VGA devices implemented on PCI will typically have the same “hard-decoded” addresses as they did on ISA. For more details see “PCI Bus Binding to IEEE Std 1275-1994 Standard for Boot (Initialization Configuration) Firmware Revision 2.1” Section 7, Legacy Devices.

The Resource Access Control (RAC) module inside the X server [0] existed for the legacy VGA arbitration task (besides other bus management tasks) when more than one legacy device co-exists on the same machine. But the problem happens when these devices are trying to be accessed by different userspace clients (e.g. two server in parallel). Their address assignments conflict. Moreover, ideally, being a userspace application, it is not the role of the X server to control bus resources. Therefore an arbitration scheme outside of the X server is needed to control the sharing of these resources. This document introduces the operation of the VGA arbiter implemented for the Linux kernel.

13.1 vgaarb kernel/userspace ABI

The vgaarb is a module of the Linux Kernel. When it is initially loaded, it scans all PCI devices and adds the VGA ones inside the arbitration. The arbiter then enables/disables the decoding on different devices of the VGA legacy instructions. Devices which do not want/need to use the arbiter may explicitly tell it by calling `vga_set_legacy_decoding()`.

The kernel exports a char device interface (`/dev/vga_arbiter`) to the clients, which has the following semantics:

open

Opens a user instance of the arbiter. By default, it’s attached to the default VGA device of the system.

close

Close a user instance. Release locks made by the user

read

Return a string indicating the status of the target like:

`"<card_ID>,decodes=<io_state>,owns=<io_state>,locks=<io_state> (ic,mc)"`

An IO state string is of the form `{io,mem,io+mem,none}`, mc and ic are respectively mem and io lock counts (for debugging/ diagnostic only). “decodes” indicate what the card currently decodes, “owns” indicates what is currently enabled on it, and “locks” indicates

what is locked by this card. If the card is unplugged, we get “invalid” then for `card_ID` and an `-ENODEV` error is returned for any command until a new card is targeted.

write

Write a command to the arbiter. List of commands:

target <card_ID>

switch target to card `<card_ID>` (see below)

lock <io_state>

acquires locks on target (“none” is an invalid `io_state`)

trylock <io_state>

non-blocking acquire locks on target (returns `EBUSY` if unsuccessful)

unlock <io_state>

release locks on target

unlock all

release all locks on target held by this user (not implemented yet)

decodes <io_state>

set the legacy decoding attributes for the card

poll

event if something changes on any card (not just the target)

`card_ID` is of the form “PCI:domain:bus:dev.fn”. It can be set to “default” to go back to the system default card (TODO: not implemented yet). Currently, only PCI is supported as a prefix, but the userland API may support other bus types in the future, even if the current kernel implementation doesn’t.

Note about locks:

The driver keeps track of which user has which locks on which card. It supports stacking, like the kernel one. This complexifies the implementation a bit, but makes the arbiter more tolerant to user space problems and able to properly cleanup in all cases when a process dies. Currently, a max of 16 cards can have locks simultaneously issued from user space for a given user (file descriptor instance) of the arbiter.

In the case of devices hot-{un,}plugged, there is a hook - `pci_notify()` - to notify them being added/removed in the system and automatically added/removed in the arbiter.

There is also an in-kernel API of the arbiter in case DRM, vgacon, or other drivers want to use it.

13.2 In-kernel interface

```
int vga_get_interruptible(struct pci_dev *pdev, unsigned int rsrc)
```

Parameters

struct pci_dev *pdev

pci device of the VGA card or NULL for the system default

unsigned int rsrc

bit mask of resources to acquire and lock

Description

Shortcut to `vga_get` with interruptible set to true.

On success, release the VGA resource again with `vga_put()`.

int **vga_get_uninterruptible**(struct pci_dev *pdev, unsigned int rsrc)
 shortcut to `vga_get()`

Parameters

struct pci_dev *pdev
 pci device of the VGA card or NULL for the system default

unsigned int rsrc
 bit mask of resources to acquire and lock

Description

Shortcut to `vga_get` with interruptible set to false.

On success, release the VGA resource again with `vga_put()`.

struct pci_dev ***vga_default_device**(void)
 return the default VGA device, for vgacon

Parameters

void
 no arguments

Description

This can be defined by the platform. The default implementation is rather dumb and will probably only work properly on single VGA card setups and/or x86 platforms.

If your VGA default device is not PCI, you'll have to return NULL here. In this case, I assume it will not conflict with any PCI card. If this is not true, I'll have to define two arch hooks for enabling/disabling the VGA default device if that is possible. This may be a problem with real `_ISA_VGA` cards, in addition to a PCI one. I don't know at this point how to deal with that card. Can their IOs be disabled at all? If not, then I suppose it's a matter of having the proper arch hook telling us about it, so we basically never allow anybody to succeed a `vga_get()`.

int **vga_remove_vgacon**(struct pci_dev *pdev)
 deactivate VGA console

Parameters

struct pci_dev *pdev
 PCI device.

Description

Unbind and unregister vgacon in case pdev is the default VGA device. Can be called by GPU drivers on initialization to make sure VGA register access done by vgacon will not disturb the device.

int **vga_get**(struct pci_dev *pdev, unsigned int rsrc, int interruptible)
 acquire & lock VGA resources

Parameters

struct pci_dev *pdev

PCI device of the VGA card or NULL for the system default

unsigned int rsrc

bit mask of resources to acquire and lock

int interruptible

blocking should be interruptible by signals ?

Description

Acquire VGA resources for the given card and mark those resources locked. If the resources requested are “normal” (and not legacy) resources, the arbiter will first check whether the card is doing legacy decoding for that type of resource. If yes, the lock is “converted” into a legacy resource lock.

The arbiter will first look for all VGA cards that might conflict and disable their IOs and/or Memory access, including VGA forwarding on P2P bridges if necessary, so that the requested resources can be used. Then, the card is marked as locking these resources and the IO and/or Memory accesses are enabled on the card (including VGA forwarding on parent P2P bridges if any).

This function will block if some conflicting card is already locking one of the required resources (or any resource on a different bus segment, since P2P bridges don’t differentiate VGA memory and IO afaik). You can indicate whether this blocking should be interruptible by a signal (for userland interface) or not.

Must not be called at interrupt time or in atomic context. If the card already owns the resources, the function succeeds. Nested calls are supported (a per-resource counter is maintained)

On success, release the VGA resource again with [vga_put\(\)](#).

0 on success, negative error code on failure.

Return

void **vga_put**(struct pci_dev *pdev, unsigned int rsrc)

release lock on legacy VGA resources

Parameters

struct pci_dev *pdev

PCI device of VGA card or NULL for system default

unsigned int rsrc

bit mask of resource to release

Description

Release resources previously locked by [vga_get\(\)](#) or [vga_tryget\(\)](#). The resources aren’t disabled right away, so that a subsequent [vga_get\(\)](#) on the same card will succeed immediately. Resources have a counter, so locks are only released if the counter reaches 0.

void **vga_set_legacy_decoding**(struct pci_dev *pdev, unsigned int decodes)

Parameters

struct pci_dev *pdev

PCI device of the VGA card

unsigned int decodes

bit mask of what legacy regions the card decodes

Description

Indicate to the arbiter if the card decodes legacy VGA IOs, legacy VGA Memory, both, or none. All cards default to both, the card driver (fbdev for example) should tell the arbiter if it has disabled legacy decoding, so the card can be left out of the arbitration process (and can be safe to take interrupts at any time).

```
int vga_client_register(struct pci_dev *pdev, unsigned int (*set_decode)(struct pci_dev
                                                                    *pdev, bool decode))
```

register or unregister a VGA arbitration client

Parameters

struct pci_dev *pdev

PCI device of the VGA client

unsigned int (*set_decode)(struct pci_dev *pdev, bool decode)

VGA decode change callback

Description

Clients have two callback mechanisms they can use.

set_decode callback: If a client can disable its GPU VGA resource, it will get a callback from this to set the encode/decode state.

Rationale: we cannot disable VGA decode resources unconditionally because some single GPU laptops seem to require ACPI or BIOS access to the VGA registers to control things like back-lights etc. Hopefully newer multi-GPU laptops do something saner, and desktops won't have any special ACPI for this. The driver will get a callback when VGA arbitration is first used by userspace since some older X servers have issues.

Does not check whether a client for **pdev** has been registered already.

To unregister, call `vga_client_unregister()`.

Return

0 on success, -ENODEV on failure

13.3 libpciaccess

To use the vga arbiter char device it was implemented an API inside the libpciaccess library. One field was added to struct pci_device (each device on the system):

```
/* the type of resource decoded by the device */
int vgaarb_rsrc;
```

Besides it, in pci_system were added:

```
int vgaarb_fd;
int vga_count;
struct pci_device *vga_target;
struct pci_device *vga_default_dev;
```

The `vga_count` is used to track how many cards are being arbitrated, so for instance, if there is only one card, then it can completely escape arbitration.

These functions below acquire VGA resources for the given card and mark those resources as locked. If the resources requested are “normal” (and not legacy) resources, the arbiter will first check whether the card is doing legacy decoding for that type of resource. If yes, the lock is “converted” into a legacy resource lock. The arbiter will first look for all VGA cards that might conflict and disable their IOs and/or Memory access, including VGA forwarding on P2P bridges if necessary, so that the requested resources can be used. Then, the card is marked as locking these resources and the IO and/or Memory access is enabled on the card (including VGA forwarding on parent P2P bridges if any). In the case of `vga_arb_lock()`, the function will block if some conflicting card is already locking one of the required resources (or any resource on a different bus segment, since P2P bridges don’t differentiate VGA memory and IO afaik). If the card already owns the resources, the function succeeds. `vga_arb_trylock()` will return `(-EBUSY)` instead of blocking. Nested calls are supported (a per-resource counter is maintained).

Set the target device of this client.

```
int pci_device_vgaarb_set_target (struct pci_device *dev);
```

For instance, in x86 if two devices on the same bus want to lock different resources, both will succeed (lock). If devices are in different buses and trying to lock different resources, only the first who tried succeeds.

```
int pci_device_vgaarb_lock      (void);
int pci_device_vgaarb_trylock   (void);
```

Unlock resources of device.

```
int pci_device_vgaarb_unlock    (void);
```

Indicates to the arbiter if the card decodes legacy VGA IOs, legacy VGA Memory, both, or none. All cards default to both, the card driver (fbdev for example) should tell the arbiter if it has disabled legacy decoding, so the card can be left out of the arbitration process (and can be safe to take interrupts at any time).

```
int pci_device_vgaarb_decodes   (int new_vgaarb_rsrc);
```

Connects to the arbiter device, allocates the struct

```
int pci_device_vgaarb_init      (void);
```

Close the connection

```
void pci_device_vgaarb_fini     (void);
```

13.4 xf86VGAArbiter (X server implementation)

X server basically wraps all the functions that touch VGA registers somehow.

13.5 References

Benjamin Herrenschmidt (IBM?) started this work when he discussed such design with the Xorg community in 2005 [1, 2]. In the end of 2007, Paulo Zanoni and Tiago Vignatti (both of C3SL/Federal University of Paraná) proceeded his work enhancing the kernel code to adapt as a kernel module and also did the implementation of the user space side [3]. Now (2009) Tiago Vignatti and Dave Airlie finally put this work in shape and queued to Jesse Barnes' PCI tree.

- 0) <https://cgit.freedesktop.org/xorg/xserver/commit/?id=4b42448a2388d40f257774fbffdccaea87bcb>
- 1) <https://lists.freedesktop.org/archives/xorg/2005-March/006663.html>
- 2) <https://lists.freedesktop.org/archives/xorg/2005-March/006745.html>
- 3) <https://lists.freedesktop.org/archives/xorg/2007-October/029507.html>

AUTOMATED TESTING OF THE DRM SUBSYSTEM

14.1 Introduction

Making sure that changes to the core or drivers don't introduce regressions can be very time-consuming when lots of different hardware configurations need to be tested. Moreover, it isn't practical for each person interested in this testing to have to acquire and maintain what can be a considerable amount of hardware.

Also, it is desirable for developers to check for regressions in their code by themselves, instead of relying on the maintainers to find them and then reporting back.

There are facilities in gitlab.freedesktop.org to automatically test Mesa that can be used as well for testing the DRM subsystem. This document explains how people interested in testing it can use this shared infrastructure to save quite some time and effort.

14.2 Relevant files

14.2.1 `drivers/gpu/drm/ci/gitlab-ci.yml`

This is the root configuration file for GitLab CI. Among other less interesting bits, it specifies the specific version of the scripts to be used. There are some variables that can be modified to change the behavior of the pipeline:

DRM_CI_PROJECT_PATH

Repository that contains the Mesa software infrastructure for CI

DRM_CI_COMMIT_SHA

A particular revision to use from that repository

UPSTREAM_REPO

URL to git repository containing the target branch

TARGET_BRANCH

Branch to which this branch is to be merged into

IGT_VERSION

Revision of igt-gpu-tools being used, from <https://gitlab.freedesktop.org/drm/igt-gpu-tools>

14.2.2 drivers/gpu/drm/ci/testlist.txt

IGT tests to be run on all drivers (unless mentioned in a driver's *-skips.txt file, see below).

14.2.3 drivers/gpu/drm/ci/\${DRIVER_NAME}-\${HW_REVISION}-fails.txt

Lists the known failures for a given driver on a specific hardware revision.

14.2.4 drivers/gpu/drm/ci/\${DRIVER_NAME}-\${HW_REVISION}-flakes.txt

Lists the tests that for a given driver on a specific hardware revision are known to behave unreliably. These tests won't cause a job to fail regardless of the result. They will still be run.

14.2.5 drivers/gpu/drm/ci/\${DRIVER_NAME}-\${HW_REVISION}-skips.txt

Lists the tests that won't be run for a given driver on a specific hardware revision. These are usually tests that interfere with the running of the test list due to hanging the machine, causing OOM, taking too long, etc.

14.3 How to enable automated testing on your tree

1. Create a Linux tree in <https://gitlab.freedesktop.org/> if you don't have one yet
2. In your kernel repo's configuration (eg. https://gitlab.freedesktop.org/janedoe/linux/-/settings/ci_cd), change the CI/CD configuration file from .gitlab-ci.yml to drivers/gpu/drm/ci/gitlab-ci.yml.
3. Next time you push to this repository, you will see a CI pipeline being created (eg. <https://gitlab.freedesktop.org/janedoe/linux/-/pipelines>)
4. The various jobs will be run and when the pipeline is finished, all jobs should be green unless a regression has been found.

14.4 How to update test expectations

If your changes to the code fix any tests, you will have to remove one or more lines from one or more of the files in drivers/gpu/drm/ci/\${DRIVER_NAME}*_fails.txt, for each of the test platforms affected by the change.

14.5 How to expand coverage

If your code changes make it possible to run more tests (by solving reliability issues, for example), you can remove tests from the flakes and/or skips lists, and then the expected results if there are any known failures.

If there is a need for updating the version of IGT being used (maybe you have added more tests to it), update the IGT_VERSION variable at the top of the gitlab-ci.yml file.

14.6 How to test your changes to the scripts

For testing changes to the scripts in the drm-ci repo, change the DRM_CI_PROJECT_PATH and DRM_CI_COMMIT_SHA variables in drivers/gpu/drm/ci/gitlab-ci.yml to match your fork of the project (eg. janedoe/drm-ci). This fork needs to be in <https://gitlab.freedesktop.org/>.

14.7 How to incorporate external fixes in your testing

Often, regressions in other trees will prevent testing changes local to the tree under test. These fixes will be automatically merged in during the build jobs from a branch in the target tree that is named as \${TARGET_BRANCH}-external-fixes.

If the pipeline is not in a merge request and a branch with the same name exists in the local tree, commits from that branch will be merged in as well.

14.8 How to deal with automated testing labs that may be down

If a hardware farm is down and thus causing pipelines to fail that would otherwise pass, one can disable all jobs that would be submitted to that farm by editing the file at <https://gitlab.freedesktop.org/gfx-ci/lab-status/-/blob/main/lab-status.yml>.

TODO LIST

This section contains a list of smaller janitorial tasks in the kernel DRM graphics subsystem useful as newbie projects. Or for slow rainy days.

15.1 Difficulty

To make it easier task are categorized into different levels:

Starter: Good tasks to get started with the DRM subsystem.

Intermediate: Tasks which need some experience with working in the DRM subsystem, or some specific GPU/display graphics knowledge. For debugging issue it's good to have the relevant hardware (or a virtual driver set up) available for testing.

Advanced: Tricky tasks that need fairly good understanding of the DRM subsystem and graphics topics. Generally need the relevant hardware for development and testing.

Expert: Only attempt these if you've successfully completed some tricky refactorings already and are an expert in the specific area

15.1.1 Subsystem-wide refactorings

15.2 Remove custom `dumb_map_offset` implementations

All GEM based drivers should be using `drm_gem_create_mmap_offset()` instead. Audit each individual driver, make sure it'll work with the generic implementation (there's lots of outdated locking leftovers in various implementations), and then remove it.

Contact: Daniel Vetter, respective driver maintainers

Level: Intermediate

15.3 Convert existing KMS drivers to atomic modesetting

3.19 has the atomic modeset interfaces and helpers, so drivers can now be converted over. Modern compositors like Wayland or Surfaceflinger on Android really want an atomic modeset interface, so this is all about the bright future.

There is a conversion guide for atomic¹ and all you need is a GPU for a non-converted driver. The “Atomic mode setting design overview” series²³ at LWN.net can also be helpful.

As part of this drivers also need to convert to universal plane (which means exposing primary & cursor as proper plane objects). But that’s much easier to do by directly using the new atomic helper driver callbacks.

Contact: Daniel Vetter, respective driver maintainers

Level: Advanced

15.4 Clean up the clipped coordination confusion around planes

We have a helper to get this right with `drm_plane_helper_check_update()`, but it’s not consistently used. This should be fixed, preferably in the atomic helpers (and drivers then moved over to clipped coordinates). Probably the helper should also be moved from `drm_plane_helper.c` to the atomic helpers, to avoid confusion - the other helpers in that file are all deprecated legacy helpers.

Contact: Ville Syrjälä, Daniel Vetter, driver maintainers

Level: Advanced

15.5 Improve plane atomic_check helpers

Aside from the clipped coordinates right above there’s a few suboptimal things with the current helpers:

- `drm_plane_helper_funcs->atomic_check` gets called for enabled or disabled planes. At best this seems to confuse drivers, worst it means they blow up when the plane is disabled without the CRTC. The only special handling is resetting values in the plane state structures, which instead should be moved into the `drm_plane_funcs->atomic_duplicate_state` functions.
- Once that’s done, helpers could stop calling `->atomic_check` for disabled planes.
- Then we could go through all the drivers and remove the more-or-less confused checks for `plane_state->fb` and `plane_state->crtc`.

Contact: Daniel Vetter

Level: Advanced

¹ <https://blog.ffwll.ch/2014/11/atomic-modeset-support-for-kms-drivers.html>

² <https://lwn.net/Articles/653071/>

³ <https://lwn.net/Articles/653466/>

15.6 Convert early atomic drivers to async commit helpers

For the first year the atomic modeset helpers didn't support asynchronous / nonblocking commits, and every driver had to hand-roll them. This is fixed now, but there's still a pile of existing drivers that easily could be converted over to the new infrastructure.

One issue with the helpers is that they require that drivers handle completion events for atomic commits correctly. But fixing these bugs is good anyway.

Somewhat related is the `legacy_cursor_update` hack, which should be replaced with the new `atomic_async_check/commit` functionality in the helpers in drivers that still look at that flag.

Contact: Daniel Vetter, respective driver maintainers

Level: Advanced

15.7 Fallout from atomic KMS

`drm_atomic_helper.c` provides a batch of functions which implement legacy IOCTLs on top of the new atomic driver interface. Which is really nice for gradual conversion of drivers, but unfortunately the semantic mismatches are a bit too severe. So there's some follow-up work to adjust the function interfaces to fix these issues:

- atomic needs the lock acquire context. At the moment that's passed around implicitly with some horrible hacks, and it's also allocate with `GFP_NOFAIL` behind the scenes. All legacy paths need to start allocating the acquire context explicitly on stack and then also pass it down into drivers explicitly so that the legacy-on-atomic functions can use them.

Except for some driver code this is done. This task should be finished by adding `WARN_ON(!drm_drv_uses_atomic_modeset)` in `drm_modeset_lock_all()`.

- A bunch of the vtable hooks are now in the wrong place: DRM has a split between core vfunc tables (named `drm_foo_funcs`), which are used to implement the userspace ABI. And then there's the optional hooks for the helper libraries (name `drm_foo_helper_funcs`), which are purely for internal use. Some of these hooks should be move from `_funcs` to `_helper_funcs` since they are not part of the core ABI. There's a `FIXME` comment in the `kernel-doc` for each such case in `drm_crtc.h`.

Contact: Daniel Vetter

Level: Intermediate

15.8 Get rid of dev->struct_mutex from GEM drivers

`dev->struct_mutex` is the Big DRM Lock from legacy days and infested everything. Nowadays in modern drivers the only bit where it's mandatory is serializing GEM buffer object destruction. Which unfortunately means drivers have to keep track of that lock and either call `unreference` or `unreference_locked` depending upon context.

Core GEM doesn't have a need for `struct_mutex` any more since kernel 4.8, and there's a GEM object free callback for any drivers which are entirely `struct_mutex` free.

For drivers that need `struct_mutex` it should be replaced with a driver- private lock. The tricky part is the BO free functions, since those can't reliably take that lock any more. Instead state needs to be protected with suitable subordinate locks or some cleanup work pushed to a worker thread. For performance-critical drivers it might also be better to go with a more fine-grained per-buffer object and per-context lockings scheme. Currently only the `msm` and `i915` drivers use `struct_mutex`.

Contact: Daniel Vetter, respective driver maintainers

Level: Advanced

15.9 Move Buffer Object Locking to `dma_resv_lock()`

Many drivers have their own per-object locking scheme, usually using `mutex_lock()`. This causes all kinds of trouble for buffer sharing, since depending which driver is the exporter and importer, the locking hierarchy is reversed.

To solve this we need one standard per-object locking mechanism, which is `dma_resv_lock()`. This lock needs to be called as the outermost lock, with all other driver specific per-object locks removed. The problem is that rolling out the actual change to the locking contract is a flag day, due to `struct dma_buf` buffer sharing.

Level: Expert

15.10 Convert logging to `drm_*` functions with `drm_device` parameter

For drivers which could have multiple instances, it is necessary to differentiate between which is which in the logs. Since `DRM_INFO/WARN/ERROR` don't do this, drivers used `dev_info/warn/err` to make this differentiation. We now have `drm_*` variants of the `drm` print functions, so we can start to convert those drivers back to using `drm`-formatted specific log messages.

Before you start this conversion please contact the relevant maintainers to make sure your work will be merged - not everyone agrees that the `DRM dmesg` macros are better.

Contact: Sean Paul, Maintainer of the driver you plan to convert

Level: Starter

15.11 Convert drivers to use simple modeset suspend/resume

Most drivers (except `i915` and `nouveau`) that use `drm_atomic_helper_suspend/resume()` can probably be converted to use `drm_mode_config_helper_suspend/resume()`. Also there's still open-coded version of the atomic suspend/resume code in older atomic modeset drivers.

Contact: Maintainer of the driver you plan to convert

Level: Intermediate

15.12 Convert drivers to use `drm_fbdev_generic_setup()`

Most drivers can use `drm_fbdev_generic_setup()`. Driver have to implement atomic modesetting and GEM vmap support. Historically, generic fbdev emulation expected the framebuffer in system memory or system-like memory. By employing struct `iosys_map`, drivers with framebuffers in I/O memory can be supported as well.

Contact: Maintainer of the driver you plan to convert

Level: Intermediate

15.13 Reimplement functions in `drm_fbdev_fb_ops` without fbdev

A number of callback functions in `drm_fbdev_fb_ops` could benefit from being rewritten without dependencies on the fbdev module. Some of the helpers could further benefit from using struct `iosys_map` instead of raw pointers.

Contact: Thomas Zimmermann <tzimmermann@suse.de>, Daniel Vetter

Level: Advanced

15.14 Benchmark and optimize blitting and format-conversion function

Drawing to display memory quickly is crucial for many applications' performance.

On at least x86-64, `sys_imageblit()` is significantly slower than `cfb_imageblit()`, even though both use the same blitting algorithm and the latter is written for I/O memory. It turns out that `cfb_imageblit()` uses `movl` instructions, while `sys_imageblit` apparently does not. This seems to be a problem with gcc's optimizer. DRM's format-conversion helpers might be subject to similar issues.

Benchmark and optimize fbdev's `sys_()` helpers and DRM's format-conversion helpers. In cases that can be further optimized, maybe implement a different algorithm. For micro-optimizations, use `movl/movq` instructions explicitly. That might possibly require architecture-specific helpers (e.g., `storel()` `storeq()`).

Contact: Thomas Zimmermann <tzimmermann@suse.de>

Level: Intermediate

15.15 `drm_framebuffer_funcs` and `drm_mode_config_funcs.fb_create` cleanup

A lot more drivers could be switched over to the `drm_gem_framebuffer` helpers. Various hold-ups:

- Need to switch over to the generic dirty tracking code using `drm_atomic_helper_dirtyfb` first (e.g. `qxl`).

- Need to switch to `drm_fbdev_generic_setup()`, otherwise a lot of the custom fb setup code can't be deleted.
- Need to switch to `drm_gem_fb_create()`, as now `drm_gem_fb_create()` checks for valid formats for atomic drivers.
- Many drivers subclass `drm_framebuffer`, we'd need an embedding compatible version of the various `drm_gem_fb_create` functions. Maybe called `drm_gem_fb_create/_with_dirty/_with_funcs` as needed.

Contact: Daniel Vetter

Level: Intermediate

15.16 Generic fbdev defio support

The defio support code in the fbdev core has some very specific requirements, which means drivers need to have a special framebuffer for fbdev. The main issue is that it uses some fields in struct page itself, which breaks shmem gem objects (and other things). To support defio, affected drivers require the use of a shadow buffer, which may add CPU and memory overhead.

Possible solution would be to write our own defio mmap code in the drm fbdev emulation. It would need to fully wrap the existing mmap ops, forwarding everything after it has done the write-protect/mkwrite trickery:

- In the `drm_fbdev_fb_mmap` helper, if we need defio, change the default page prots to write-protected with something like this:

```
vma->vm_page_prot = pgprot_wrprotect(vma->vm_page_prot);
```

- Set the mkwrite and fsync callbacks with similar implementations to the core fbdev defio stuff. These should all work on plain ptes, they don't actually require a struct page. uff. These should all work on plain ptes, they don't actually require a struct page.
- Track the dirty pages in a separate structure (bitfield with one bit per page should work) to avoid clobbering struct page.

Might be good to also have some igt testcases for this.

Contact: Daniel Vetter, Noralf Tronnes

Level: Advanced

15.17 connector register/unregister fixes

- For most connectors it's a no-op to call `drm_connector_register/unregister` directly from driver code, `drm_dev_register/unregister` take care of this already. We can remove all of them.
- For dp drivers it's a bit more a mess, since we need the connector to be registered when calling `drm_dp_aux_register`. Fix this by instead calling `drm_dp_aux_init`, and moving the actual registering into a `late_register` callback as recommended in the kerneldoc.

Level: Intermediate

15.18 Remove load/unload callbacks

The load/unload callbacks in struct `&drm_driver` are very much midlayers, plus for historical reasons they get the ordering wrong (and we can't fix that) between setting up the `&drm_driver` structure and calling `drm_dev_register()`.

- Rework drivers to no longer use the load/unload callbacks, directly coding the load/unload sequence into the driver's probe function.
- Once all drivers are converted, remove the load/unload callbacks.

Contact: Daniel Vetter

Level: Intermediate

15.19 Replace `drm_detect_hdmi_monitor()` with `drm_display_info.is_hdmi`

Once EDID is parsed, the monitor HDMI support information is available through `drm_display_info.is_hdmi`. Many drivers still call `drm_detect_hdmi_monitor()` to retrieve the same information, which is less efficient.

Audit each individual driver calling `drm_detect_hdmi_monitor()` and switch to `drm_display_info.is_hdmi` if applicable.

Contact: Laurent Pinchart, respective driver maintainers

Level: Intermediate

15.20 Consolidate custom driver modeset properties

Before atomic modeset took place, many drivers were creating their own properties. Among other things, atomic brought the requirement that custom, driver specific properties should not be used.

For this task, we aim to introduce core helpers or reuse the existing ones if available:

A quick, unconfirmed, examples list.

Introduce core helpers: - audio (amdgpu, intel, gma500, radeon) - brightness, contrast, etc (armada, nouveau) - overlay only (?) - broadcast rgb (gma500, intel) - colorkey (armada, nouveau, rcar) - overlay only (?) - dither (amdgpu, nouveau, radeon) - varies across drivers - underscan family (amdgpu, radeon, nouveau)

Already in core: - colorspace (sti) - tv format names, enhancements (gma500, intel) - tv overscan, margins, etc. (gma500, intel) - zorder (omapdrm) - same as zpos (?)

Contact: Emil Velikov, respective driver maintainers

Level: Intermediate

15.21 Use struct iosys_map throughout codebase

Pointers to shared device memory are stored in struct iosys_map. Each instance knows whether it refers to system or I/O memory. Most of the DRM-wide interface have been converted to use struct iosys_map, but implementations often still use raw pointers.

The task is to use struct iosys_map where it makes sense.

- Memory managers should use struct iosys_map for dma-buf-imported buffers.
- TTM might benefit from using struct iosys_map internally.
- Framebuffer copying and blitting helpers should operate on struct iosys_map.

Contact: Thomas Zimmermann <tzimmermann@suse.de>, Christian König, Daniel Vetter

Level: Intermediate

15.22 Review all drivers for setting struct drm_mode_config.{max_width,max_height} correctly

The values in `struct drm_mode_config.{max_width,max_height}` describe the maximum supported framebuffer size. It's the virtual screen size, but many drivers treat it like limitations of the physical resolution.

The maximum width depends on the hardware's maximum scanline pitch. The maximum height depends on the amount of addressable video memory. Review all drivers to initialize the fields to the correct values.

Contact: Thomas Zimmermann <tzimmermann@suse.de>

Level: Intermediate

15.23 Request memory regions in all drivers

Go through all drivers and add code to request the memory regions that the driver uses. This requires adding calls to `request_mem_region()`, `pci_request_region()` or similar functions. Use helpers for managed cleanup where possible.

Drivers are pretty bad at doing this and there used to be conflicts among DRM and fbdev drivers. Still, it's the correct thing to do.

Contact: Thomas Zimmermann <tzimmermann@suse.de>

Level: Starter

15.24 Remove driver dependencies on FB_DEVICE

A number of fbdev drivers provide attributes via sysfs and therefore depend on CONFIG_FB_DEVICE to be selected. Review each driver and attempt to make any dependencies on CONFIG_FB_DEVICE optional. At the minimum, the respective code in the driver could be conditionalized via `ifdef CONFIG_FB_DEVICE`. Not all drivers might be able to drop CONFIG_FB_DEVICE.

Contact: Thomas Zimmermann <tzimmermann@suse.de>

Level: Starter

15.25 Clean up checks for already prepared/enabled in panels

In a whole pile of panel drivers, we have code to make the prepare/unprepare/enable/disable callbacks behave as no-ops if they've already been called. To get some idea of the duplicated code, try:

```
git grep 'if.*>prepared' -- drivers/gpu/drm/panel
git grep 'if.*>enabled' -- drivers/gpu/drm/panel
```

In the patch (“drm/panel: Check for already prepared/enabled in drm_panel”) we’ve moved this check to the core. Now we can most definitely remove the check from the individual panels and save a pile of code.

In addition to removing the check from the individual panels, it is believed that even the core shouldn’t need this check and that should be considered an error if other code ever relies on this check. The check in the core currently prints a warning whenever something is relying on this check with `dev_warn()`. After a little while, we likely want to promote this to a `WARN(1)` to help encourage folks not to rely on this behavior.

Contact: Douglas Anderson <dianders@chromium.org>

Level: Starter/Intermediate

15.25.1 Core refactorings

15.26 Make panic handling work

This is a really varied tasks with lots of little bits and pieces:

- The panic path can’t be tested currently, leading to constant breaking. The main issue here is that panics can be triggered from hardirq contexts and hence all panic related callback can run in hardirq context. It would be awesome if we could test at least the fbdev helper code and driver code by e.g. trigger calls through drm debugfs files. hardirq context could be achieved by using an IPI to the local processor.
- There’s a massive confusion of different panic handlers. DRM fbdev emulation helpers had their own (long removed), but on top of that the fbcon code itself also has one. We need to make sure that they stop fighting over each other. This is worked around by checking `oops_in_progress` at various entry points into the DRM fbdev emulation helpers. A much cleaner approach here would be to switch fbcon to the [threaded printk support](#).

- `drm_can_sleep()` is a mess. It hides real bugs in normal operations and isn't a full solution for panic paths. We need to make sure that it only returns true if there's a panic going on for real, and fix up all the fallout.
- The panic handler must never sleep, which also means it can't ever `mutex_lock()`. Also it can't grab any other lock unconditionally, not even spinlocks (because NMI and hardirq can panic too). We need to either make sure to not call such paths, or trylock everything. Really tricky.
- A clean solution would be an entirely separate panic output support in KMS, bypassing the current fbcon support. See [\[PATCH v2 0/3\] drm: Add panic handling](#).
- Encoding the actual oops and preceding dmesg in a QR might help with the dread "important stuff scrolled away" problem. See [\[RFC\]\[PATCH\] Oops messages transfer using QR codes](#) for some example code that could be reused.

Contact: Daniel Vetter

Level: Advanced

15.27 Clean up the debugfs support

There's a bunch of issues with it:

- Convert drivers to support the `drm_debugfs_add_files()` function instead of the `drm_debugfs_create_files()` function.
- Improve late-register debugfs by rolling out the same debugfs pre-register infrastructure for connector and crtc too. That way, the drivers won't need to split their setup code into init and register anymore.
- We probably want to have some support for debugfs files on crtc/connectors and maybe other kms objects directly in core. There's even `drm_print` support in the funcs for these objects to dump kms state, so it's all there. And then the `->show()` functions should obviously give you a pointer to the right object.
- The `drm_driver->debugfs_init` hooks we have is just an artifact of the old midlayered load sequence. DRM debugfs should work more like sysfs, where you can create properties/files for an object anytime you want, and the core takes care of publishing/unpublishing all the files at register/unregister time. Drivers shouldn't need to worry about these technicalities, and fixing this (together with the `drm_minor->drm_device` move) would allow us to remove `debugfs_init`.

Contact: Daniel Vetter

Level: Intermediate

15.28 Object lifetime fixes

There's two related issues here

- Cleanup up the various `->destroy` callbacks, which often are all the same simple code.
- Lots of drivers erroneously allocate DRM modeset objects using `devm_kzalloc`, which results in use-after free issues on driver unload. This can be serious trouble even for drivers for hardware integrated on the SoC due to `EPROBE_DEFERRED` backoff.

Both these problems can be solved by switching over to `drm_kzalloc()`, and the various convenience wrappers provided, e.g. `drm_crtc_alloc_with_planes()`, `drm_universal_plane_alloc()`, ... and so on.

Contact: Daniel Vetter

Level: Intermediate

15.29 Remove automatic page mapping from dma-buf importing

When importing dma-bufs, the dma-buf and PRIME frameworks automatically map imported pages into the importer's DMA area. `drm_gem_prime_fd_to_handle()` and `drm_gem_prime_handle_to_fd()` require that importers call `dma_buf_attach()` even if they never do actual device DMA, but only CPU access through `dma_buf_vmap()`. This is a problem for USB devices, which do not support DMA operations.

To fix the issue, automatic page mappings should be removed from the buffer-sharing code. Fixing this is a bit more involved, since the import/export cache is also tied to `&drm_gem_object.import_attach`. Meanwhile we paper over this problem for USB devices by fishing out the USB host controller device, as long as that supports DMA. Otherwise importing can still needlessly fail.

Contact: Thomas Zimmermann <tzimmermann@suse.de>, Daniel Vetter

Level: Advanced

15.29.1 Better Testing

15.30 Add unit tests using the Kernel Unit Testing (KUnit) framework

The `KUnit` provides a common framework for unit tests within the Linux kernel. Having a test suite would allow to identify regressions earlier.

A good candidate for the first unit tests are the format-conversion helpers in `drm_format_helper.c`.

Contact: Javier Martinez Canillas <javierm@redhat.com>

Level: Intermediate

15.31 Enable trinity for DRM

And fix up the fallout. Should be really interesting ...

Level: Advanced

15.32 Make KMS tests in i-g-t generic

The i915 driver team maintains an extensive testsuite for the i915 DRM driver, including tons of testcases for corner-cases in the modesetting API. It would be awesome if those tests (at least the ones not relying on Intel-specific GEM features) could be made to run on any KMS driver.

Basic work to run i-g-t tests on non-i915 is done, what's now missing is mass- converting things over. For modeset tests we also first need a bit of infrastructure to use dumb buffers for untiled buffers, to be able to run all the non-i915 specific modeset tests.

Level: Advanced

15.33 Extend virtual test driver (VKMS)

See the documentation of [VKMS](#) for more details. This is an ideal internship task, since it only requires a virtual machine and can be sized to fit the available time.

Level: See details

15.34 Backlight Refactoring

Backlight drivers have a triple enable/disable state, which is a bit overkill. Plan to fix this:

1. Roll out `backlight_enable()` and `backlight_disable()` helpers everywhere. This has started already.
2. In all, only look at one of the three status bits set by the above helpers.
3. Remove the other two status bits.

Contact: Daniel Vetter

Level: Intermediate

15.34.1 Driver Specific

15.35 AMD DC Display Driver

AMD DC is the display driver for AMD devices starting with Vega. There has been a bunch of progress cleaning it up but there's still plenty of work to be done.

See `drivers/gpu/drm/amd/display/TODO` for tasks.

Contact: Harry Wentland, Alex Deucher

15.35.1 Bootsplash

There is support in place now for writing internal DRM clients making it possible to pick up the bootsplash work that was rejected because it was written for fbdev.

- [v6,8/8] drm/client: Hack: Add bootsplash example <https://patchwork.freedesktop.org/patch/306579/>
- [RFC PATCH v2 00/13] Kernel based bootsplash <https://lore.kernel.org/r/20171213194755.3409-1-mstaudt@suse.de>

Contact: Sam Ravnborg

Level: Advanced

15.35.2 Brightness handling on devices with multiple internal panels

On x86/ACPI devices there can be multiple backlight firmware interfaces: (ACPI) video, vendor specific and others. As well as direct/native (PWM) register programming by the KMS driver.

To deal with this backlight drivers used on x86/ACPI call `acpi_video_get_backlight_type()` which has heuristics (+quirks) to select which backlight interface to use; and backlight drivers which do not match the returned type will not register themselves, so that only one backlight device gets registered (in a single GPU setup, see below).

At the moment this more or less assumes that there will only be 1 (internal) panel on a system.

On systems with 2 panels this may be a problem, depending on what interface `acpi_video_get_backlight_type()` selects:

1. native: in this case the KMS driver is expected to know which backlight device belongs to which output so everything should just work.
2. video: this does support controlling multiple backlights, but some work will need to be done to get the output <-> backlight device mapping

The above assumes both panels will require the same backlight interface type. Things will break on systems with multiple panels where the 2 panels need a different type of control. E.g. one panel needs ACPI video backlight control, where as the other is using native backlight control. Currently in this case only one of the 2 required backlight devices will get registered, based on the `acpi_video_get_backlight_type()` return value.

If this (theoretical) case ever shows up, then supporting this will need some work. A possible solution here would be to pass a device and connector-name to `acpi_video_get_backlight_type()` so that it can deal with this.

Note in a way we already have a case where userspace sees 2 panels, in dual GPU laptop setups with a mux. On those systems we may see either 2 native backlight devices; or 2 native backlight devices.

Userspace already has code to deal with this by detecting if the related panel is active (iow which way the mux between the GPU and the panels points) and then uses that backlight device. Userspace here very much assumes a single panel though. It picks only 1 of the 2 backlight devices and then only uses that one.

Note that all userspace code (that I know off) is currently hardcoded to assume a single panel.

Before the recent changes to not register multiple (e.g. video + native) `/sys/class/backlight` devices for a single panel (on a single GPU laptop), userspace would see multiple backlight devices all controlling the same backlight.

To deal with this userspace had to always pick one preferred device under `/sys/class/backlight` and will ignore the others. So to support brightness control on multiple panels userspace will need to be updated too.

There are plans to allow brightness control through the KMS API by adding a “display brightness” property to `drm_connector` objects for panels. This solves a number of issues with the `/sys/class/backlight` API, including not being able to map a sysfs backlight device to a specific connector. Any userspace changes to add support for brightness control on devices with multiple panels really should build on top of this new KMS property.

Contact: Hans de Goede

Level: Advanced

15.35.3 Outside DRM

15.36 Convert fbdev drivers to DRM

There are plenty of fbdev drivers for older hardware. Some hardware has become obsolete, but some still provides good(-enough) framebuffers. The drivers that are still useful should be converted to DRM and afterwards removed from fbdev.

Very simple fbdev drivers can best be converted by starting with a new DRM driver. Simple KMS helpers and SHMEM should be able to handle any existing hardware. The new driver’s call-back functions are filled from existing fbdev code.

More complex fbdev drivers can be refactored step-by-step into a DRM driver with the help of the DRM fbconv helpers⁴. These helpers provide the transition layer between the DRM core infrastructure and the fbdev driver interface. Create a new DRM driver on top of the fbconv helpers, copy over the fbdev driver, and hook it up to the DRM code. Examples for several fbdev drivers are available in Thomas Zimmermann’s fbconv tree⁴, as well as a tutorial of this process⁵. The result is a primitive DRM driver that can run X11 and Weston.

Contact: Thomas Zimmermann <tzimmermann@suse.de>

Level: Advanced

⁴ <https://gitlab.freedesktop.org/tzimmermann/linux/tree/fbconv>

⁵ https://gitlab.freedesktop.org/tzimmermann/linux/blob/fbconv/drivers/gpu/drm/drm_fbconv_helper.c

GPU RFC SECTION

For complex work, especially new uapi, it is often good to nail the high level design issues before getting lost in the code details. This section is meant to host such documentation:

- Each RFC should be a section in this file, explaining the goal and main design considerations. Especially for uapi make sure you Cc: all relevant project mailing lists and involved people outside of dri-devel.
- For uapi structures add a file to this directory with and then pull the kerneldoc in like with real uapi headers.
- Once the code has landed move all the documentation to the right places in the main core, helper or driver sections.

16.1 I915 DG1/LMEM RFC Section

16.1.1 Upstream plan

For upstream the overall plan for landing all the DG1 stuff and turning it for real, with all the uAPI bits is:

- Merge basic HW enabling of DG1(still without pciid)
- **Merge the uAPI bits behind special CONFIG_BROKEN(or so) flag**
 - At this point we can still make changes, but importantly this lets us start running IGTs which can utilize local-memory in CI
- **Convert over to TTM, make sure it all keeps working. Some of the work items:**
 - TTM shrinker for discrete
 - dma_resv_lockitem for full dma_resv_lock, i.e not just trylock
 - Use TTM CPU pagefault handler
 - Route shmem backend over to TTM SYSTEM for discrete
 - TTM purgeable object support
 - Move i915 buddy allocator over to TTM
- Send RFC(with mesa-dev on cc) for final sign off on the uAPI
- Add pciid for DG1 and turn on uAPI for real

16.2 I915 GuC Submission/DRM Scheduler Section

16.2.1 Upstream plan

For upstream the overall plan for landing GuC submission and integrating the i915 with the DRM scheduler is:

- **Merge basic GuC submission**

- Basic submission support for all gen11+ platforms
- Not enabled by default on any current platforms but can be enabled via modparam `enable_guc`
- Lots of rework will need to be done to integrate with DRM scheduler so no need to nit pick everything in the code, it just should be functional, no major coding style / layering errors, and not regress execlists
- Update IGTs / selftests as needed to work with GuC submission
- Enable CI on supported platforms for a baseline
- Rework / get CI healthy for GuC submission in place as needed

- **Merge new parallel submission uAPI**

- Bonding uAPI completely incompatible with GuC submission, plus it has severe design issues in general, which is why we want to retire it no matter what
- New uAPI adds `I915_CONTEXT_ENGINES_EXT_PARALLEL` context setup step which configures a slot with N contexts
- After `I915_CONTEXT_ENGINES_EXT_PARALLEL` a user can submit N batches to a slot in a single `execbuf IOCTL` and the batches run on the GPU in parallel
- Initially only for GuC submission but execlists can be supported if needed

- **Convert the i915 to use the DRM scheduler**

- **GuC submission backend fully integrated with DRM scheduler**

- * All request queues removed from backend (e.g. all backpressure handled in DRM scheduler)
- * Resets / cancels hook in DRM scheduler
- * Watchdog hooks into DRM scheduler
- * Lots of complexity of the GuC backend can be pulled out once integrated with DRM scheduler (e.g. state machine gets simpler, locking gets simpler, etc...)

- **Execlists backend will minimum required to hook in the DRM scheduler**

- * Legacy interface
- * Features like timeslicing / preemption / virtual engines would be difficult to integrate with the DRM scheduler and these features are not required for GuC submission as the GuC does these things for us
- * ROI low on fully integrating into DRM scheduler
- * Fully integrating would add lots of complexity to DRM scheduler

- **Port i915 priority inheritance / boosting feature in DRM scheduler**
 - * Used for i915 page flip, may be useful to other DRM drivers as well
 - * Will be an optional feature in the DRM scheduler
- **Remove in-order completion assumptions from DRM scheduler**
 - * Even when using the DRM scheduler the backends will handle preemption, timeslicing, etc... so it is possible for jobs to finish out of order
- Pull out i915 priority levels and use DRM priority levels
- Optimize DRM scheduler as needed

16.2.2 TODOs for GuC submission upstream

- Need an update to GuC firmware / i915 to enable error state capture
- Open source tool to decode GuC logs
- Public GuC spec

16.2.3 New uAPI for basic GuC submission

No major changes are required to the uAPI for basic GuC submission. The only change is a new scheduler attribute: `I915_SCHEDULER_CAP_STATIC_PRIORITY_MAP`. This attribute indicates the 2k i915 user priority levels are statically mapped into 3 levels as follows:

- -1k to -1 Low priority
- 0 Medium priority
- 1 to 1k High priority

This is needed because the GuC only has 4 priority bands. The highest priority band is reserved with the kernel. This aligns with the DRM scheduler priority levels too.

Spec references:

- https://www.khronos.org/registry/EGL/extensions/IMG/EGL_IMG_context_priority.txt
- <https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/chap5.html#devsandqueues-priority>
- <https://spec.oneapi.com/level-zero/latest/core/api.html#ze-command-queue-priority-t>

16.2.4 New parallel submission uAPI

The existing bonding uAPI is completely broken with GuC submission because whether a submission is a single context submit or parallel submit isn't known until execbuf time activated via the I915_SUBMIT_FENCE. To submit multiple contexts in parallel with the GuC the context must be explicitly registered with N contexts and all N contexts must be submitted in a single command to the GuC. The GuC interfaces do not support dynamically changing between N contexts as the bonding uAPI does. Hence the need for a new parallel submission interface. Also the legacy bonding uAPI is quite confusing and not intuitive at all. Furthermore I915_SUBMIT_FENCE is by design a future fence, so not really something we should continue to support.

The new parallel submission uAPI consists of 3 parts:

- Export engines logical mapping
- A 'set_parallel' extension to configure contexts for parallel submission
- Extend execbuf2 IOCTL to support submitting N BBs in a single IOCTL

Export engines logical mapping

Certain use cases require BBs to be placed on engine instances in logical order (e.g. split-frame on gen11+). The logical mapping of engine instances can change based on fusing. Rather than making UMDs be aware of fusing, simply expose the logical mapping with the existing query engine info IOCTL. Also the GuC submission interface currently only supports submitting multiple contexts to engines in logical order which is a new requirement compared to execlists. Lastly, all current platforms have at most 2 engine instances and the logical order is the same as uAPI order. This will change on platforms with more than 2 engine instances.

A single bit will be added to `drm_i915_engine_info.flags` indicating that the logical instance has been returned and a new field, `drm_i915_engine_info.logical_instance`, returns the logical instance.

A 'set_parallel' extension to configure contexts for parallel submission

The 'set_parallel' extension configures a slot for parallel submission of N BBs. It is a setup step that must be called before using any of the contexts. See `I915_CONTEXT_ENGINES_EXT_LOAD_BALANCE` or `I915_CONTEXT_ENGINES_EXT_BOND` for similar existing examples. Once a slot is configured for parallel submission the `execbuf2` IOCTL can be called submitting N BBs in a single IOCTL. Initially only supports GuC submission. Execlists supports can be added later if needed.

Add `I915_CONTEXT_ENGINES_EXT_PARALLEL_SUBMIT` and `drm_i915_context_engines_parallel_submit` to the uAPI to implement this extension.

struct **i915_context_engines_parallel_submit**

Configure engine for parallel submission.

Definition:

```
struct i915_context_engines_parallel_submit {
    struct i915_user_extension base;
    __u16 engine_index;
```

```

__u16 width;
__u16 num_siblings;
__u16 mbz16;
__u64 flags;
__u64 mbz64[3];
struct i915_engine_class_instance engines[];
};

```

Members

base

base user extension.

engine_index

slot for parallel engine

width

number of contexts per parallel engine or in other words the number of batches in each submission

num_siblings

number of siblings per context or in other words the number of possible placements for each submission

mbz16

reserved for future use; must be zero

flags

all undefined flags must be zero, currently not defined flags

mbz64

reserved for future use; must be zero

engines

2-d array of engine instances to configure parallel engine

length = width (i) * num_siblings (j) index = j + i * num_siblings

Description

Setup a slot in the context engine map to allow multiple BBs to be submitted in a single execbuf IOCTL. Those BBs will then be scheduled to run on the GPU in parallel. Multiple hardware contexts are created internally in the i915 to run these BBs. Once a slot is configured for N BBs only N BBs can be submitted in each execbuf IOCTL and this is implicit behavior e.g. The user doesn't tell the execbuf IOCTL there are N BBs, the execbuf IOCTL knows how many BBs there are based on the slot's configuration. The N BBs are the last N buffer objects or first N if I915_EXEC_BATCH_FIRST is set.

The default placement behavior is to create implicit bonds between each context if each context maps to more than 1 physical engine (e.g. context is a virtual engine). Also we only allow contexts of same engine class and these contexts must be in logically contiguous order. Examples of the placement behavior are described below. Lastly, the default is to not allow BBs to be preempted mid-batch. Rather insert coordinated preemption points on all hardware contexts between each set of BBs. Flags could be added in the future to change both of these default behaviors.

Returns -EINVAL if hardware context placement configuration is invalid or if the placement configuration isn't supported on the platform / submission interface. Returns -ENODEV if extension isn't supported on the platform / submission interface.

Examples syntax:

```
CS[X] = generic engine of same class, logical instance X
INVALID = I915_ENGINE_CLASS_INVALID, I915_ENGINE_CLASS_INVALID_NONE
```

Example 1 pseudo code:

```
set_engines(INVALID)
set_parallel(engine_index=0, width=2, num_siblings=1,
             engines=CS[0],CS[1])
```

Results in the following valid placement:

```
CS[0], CS[1]
```

Example 2 pseudo code:

```
set_engines(INVALID)
set_parallel(engine_index=0, width=2, num_siblings=2,
             engines=CS[0],CS[2],CS[1],CS[3])
```

Results in the following valid placements:

```
CS[0], CS[1]
```

```
CS[2], CS[3]
```

This can be thought of as two virtual engines, each containing two engines thereby making a 2D array. However, there are bonds tying the entries together and placing restrictions on how they can be scheduled. Specifically, the scheduler can choose only vertical columns from the 2D array. That is, CS[0] is bonded to CS[1] and CS[2] to CS[3]. So if the scheduler wants to submit to CS[0], it must also choose CS[1] and vice versa. Same for CS[2] requires also using CS[3].

```
VE[0] = CS[0], CS[2]
```

```
VE[1] = CS[1], CS[3]
```

Example 3 pseudo code:

```
set_engines(INVALID)
set_parallel(engine_index=0, width=2, num_siblings=2,
             engines=CS[0],CS[1],CS[1],CS[3])
```

Results in the following valid and invalid placements:

```
CS[0], CS[1]
```

```
CS[1], CS[3] - Not logically contiguous, return -EINVAL
```

Extend execbuf2 IOCTL to support submitting N BBs in a single IOCTL

Contexts that have been configured with the 'set_parallel' extension can only submit N BBs in a single execbuf2 IOCTL. The BBs are either the last N objects in the `drm_i915_gem_exec_object2` list or the first N if `I915_EXEC_BATCH_FIRST` is set. The number of BBs is implicit based on the slot submitted and how it has been configured by 'set_parallel' or other extensions. No uAPI changes are required to the execbuf2 IOCTL.

16.3 I915 Small BAR RFC Section

Starting from DG2 we will have resizable BAR support for device local-memory(i.e `I915_MEMORY_CLASS_DEVICE`), but in some cases the final BAR size might still be smaller than the total probed_size. In such cases, only some subset of `I915_MEMORY_CLASS_DEVICE` will be CPU accessible(for example the first 256M), while the remainder is only accessible via the GPU.

16.3.1 I915_GEM_CREATE_EXT_FLAG_NEEDS_CPU_ACCESS flag

New `gem_create_ext` flag to tell the kernel that a BO will require CPU access. This becomes important when placing an object in `I915_MEMORY_CLASS_DEVICE`, where underneath the device has a small BAR, meaning only some portion of it is CPU accessible. Without this flag the kernel will assume that CPU access is not required, and prioritize using the non-CPU visible portion of `I915_MEMORY_CLASS_DEVICE`.

struct `__drm_i915_gem_create_ext`

Existing `gem_create` behaviour, with added extension support using *struct i915_user_extension*.

Definition:

```
struct __drm_i915_gem_create_ext {
    __u64 size;
    __u32 handle;
#define I915_GEM_CREATE_EXT_FLAG_NEEDS_CPU_ACCESS (1 << 0);
    __u32 flags;
#define I915_GEM_CREATE_EXT_MEMORY_REGIONS 0;
#define I915_GEM_CREATE_EXT_PROTECTED_CONTENT 1;
    __u64 extensions;
};
```

Members

size

Requested size for the object.

The (page-aligned) allocated size for the object will be returned.

Note that for some devices we have might have further minimum page-size restrictions (larger than 4K), like for device local-memory. However in general the final size here should always reflect any rounding up, if for example using the `I915_GEM_CREATE_EXT_MEMORY_REGIONS` extension to place the object in device

local-memory. The kernel will always select the largest minimum page-size for the set of possible placements as the value to use when rounding up the **size**.

handle

Returned handle for the object.

Object handles are nonzero.

flags

Optional flags.

Supported values:

`I915_GEM_CREATE_EXT_FLAG_NEEDS_CPU_ACCESS` - Signal to the kernel that the object will need to be accessed via the CPU.

Only valid when placing objects in `I915_MEMORY_CLASS_DEVICE`, and only strictly required on configurations where some subset of the device memory is directly visible/mappable through the CPU (which we also call small BAR), like on some DG2+ systems. Note that this is quite undesirable, but due to various factors like the client CPU, BIOS etc it's something we can expect to see in the wild. See [__drm_i915_memory_region_info.probed_cpu_visible_size](#) for how to determine if this system applies.

Note that one of the placements MUST be `I915_MEMORY_CLASS_SYSTEM`, to ensure the kernel can always spill the allocation to system memory, if the object can't be allocated in the mappable part of `I915_MEMORY_CLASS_DEVICE`.

Also note that since the kernel only supports flat-CCS on objects that can *only* be placed in `I915_MEMORY_CLASS_DEVICE`, we therefore don't support `I915_GEM_CREATE_EXT_FLAG_NEEDS_CPU_ACCESS` together with flat-CCS.

Without this hint, the kernel will assume that non-mappable `I915_MEMORY_CLASS_DEVICE` is preferred for this object. Note that the kernel can still migrate the object to the mappable part, as a last resort, if userspace ever CPU faults this object, but this might be expensive, and so ideally should be avoided.

On older kernels which lack the relevant small-bar uAPI support (see also [__drm_i915_memory_region_info.probed_cpu_visible_size](#)), usage of the flag will result in an error, but it should NEVER be possible to end up with a small BAR configuration, assuming we can also successfully load the i915 kernel module. In such cases the entire `I915_MEMORY_CLASS_DEVICE` region will be CPU accessible, and as such there are zero restrictions on where the object can be placed.

extensions

The chain of extensions to apply to this object.

This will be useful in the future when we need to support several different extensions, and we need to apply more than one when creating the object. See [struct i915_user_extension](#).

If we don't supply any extensions then we get the same old `gem_create` behaviour.

For `I915_GEM_CREATE_EXT_MEMORY_REGIONS` usage see [struct drm_i915_gem_create_ext_memory_regions](#).

For `I915_GEM_CREATE_EXT_PROTECTED_CONTENT` usage see [struct drm_i915_gem_create_ext_protected_content](#).

Description

Note that new buffer flags should be added here, at least for the stuff that is immutable. Previously we would have two ioctls, one to create the object with `gem_create`, and another to apply various parameters, however this creates some ambiguity for the params which are considered immutable. Also in general we're phasing out the various SET/GET ioctls.

16.3.2 probed_cpu_visible_size attribute

New `struct __drm_i915_memory_region` attribute which returns the total size of the CPU accessible portion, for the particular region. This should only be applicable for `I915_MEMORY_CLASS_DEVICE`. We also report the `unallocated_cpu_visible_size`, alongside the `unallocated_size`.

Vulkan will need this as part of creating a separate `VkMemoryHeap` with the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` set, to represent the CPU visible portion, where the total size of the heap needs to be known. It also wants to be able to give a rough estimate of how memory can potentially be allocated.

`struct __drm_i915_memory_region_info`

Describes one region as known to the driver.

Definition:

```
struct __drm_i915_memory_region_info {
    struct drm_i915_gem_memory_class_instance region;
    __u32 rsvd0;
    __u64 probed_size;
    __u64 unallocated_size;
    union {
        __u64 rsvd1[8];
        struct {
            __u64 probed_cpu_visible_size;
            __u64 unallocated_cpu_visible_size;
        };
    };
};
```

Members

region

The class:instance pair encoding

rsvd0

MBZ

probed_size

Memory probed by the driver

Note that it should not be possible to ever encounter a zero value here, also note that no current region type will ever return -1 here. Although for future region types, this might be a possibility. The same applies to the other size fields.

unallocated_size

Estimate of memory remaining

Requires CAP_PERFMON or CAP_SYS_ADMIN to get reliable accounting. Without this (or if this is an older kernel) the value here will always equal the **probed_size**. Note this is only currently tracked for I915_MEMORY_CLASS_DEVICE regions (for other types the value here will always equal the **probed_size**).

{unnamed_union}

anonymous

rsvd1

MBZ

{unnamed_struct}

anonymous

probed_cpu_visible_size

Memory probed by the driver that is CPU accessible.

This will be always be \leq **probed_size**, and the remainder (if there is any) will not be CPU accessible.

On systems without small BAR, the **probed_size** will always equal the **probed_cpu_visible_size**, since all of it will be CPU accessible.

Note this is only tracked for I915_MEMORY_CLASS_DEVICE regions (for other types the value here will always equal the **probed_size**).

Note that if the value returned here is zero, then this must be an old kernel which lacks the relevant small-bar uAPI support (including I915_GEM_CREATE_EXT_FLAG_NEEDS_CPU_ACCESS), but on such systems we should never actually end up with a small BAR configuration, assuming we are able to load the kernel module. Hence it should be safe to treat this the same as when **probed_cpu_visible_size == probed_size**.

unallocated_cpu_visible_size

Estimate of CPU visible memory remaining

Note this is only tracked for I915_MEMORY_CLASS_DEVICE regions (for other types the value here will always equal the **probed_cpu_visible_size**).

Requires CAP_PERFMON or CAP_SYS_ADMIN to get reliable accounting. Without this the value here will always equal the **probed_cpu_visible_size**. Note this is only currently tracked for I915_MEMORY_CLASS_DEVICE regions (for other types the value here will also always equal the **probed_cpu_visible_size**).

If this is an older kernel the value here will be zero, see also **probed_cpu_visible_size**.

Description

Note this is using both *struct drm_i915_query_item* and *struct drm_i915_query*. For this new query we are adding the new query id DRM_I915_QUERY_MEMORY_REGIONS at *drm_i915_query_item.query_id*.

16.3.3 Error Capture restrictions

With error capture we have two new restrictions:

- 1) Error capture is best effort on small BAR systems; if the pages are not CPU accessible, at the time of capture, then the kernel is free to skip trying to capture them.
- 2) On discrete and newer integrated platforms we now reject error capture on recoverable contexts. In the future the kernel may want to blit during error capture, when for example something is not currently CPU accessible.

16.4 I915 VM_BIND feature design and use cases

16.4.1 VM_BIND feature

DRM_I915_GEM_VM_BIND/UNBIND ioctls allows UMD to bind/unbind GEM buffer objects (BOs) or sections of a BOs at specified GPU virtual addresses on a specified address space (VM). These mappings (also referred to as persistent mappings) will be persistent across multiple GPU submissions (execbuf calls) issued by the UMD, without user having to provide a list of all required mappings during each submission (as required by older execbuf mode).

The VM_BIND/UNBIND calls allow UMDs to request a timeline out fence for signaling the completion of bind/unbind operation.

VM_BIND feature is advertised to user via I915_PARAM_VM_BIND_VERSION. User has to opt-in for VM_BIND mode of binding for an address space (VM) during VM creation time via I915_VM_CREATE_FLAGS_USE_VM_BIND extension.

VM_BIND/UNBIND ioctl calls executed on different CPU threads concurrently are not ordered. Furthermore, parts of the VM_BIND/UNBIND operations can be done asynchronously, when valid out fence is specified.

VM_BIND features include:

- Multiple Virtual Address (VA) mappings can map to the same physical pages of an object (aliasing).
- VA mapping can map to a partial section of the BO (partial binding).
- Support capture of persistent mappings in the dump upon GPU error.
- Support for userptr gem objects (no special uapi is required for this).

TLB flush consideration

The i915 driver flushes the TLB for each submission and when an object's pages are released. The VM_BIND/UNBIND operation will not do any additional TLB flush. Any VM_BIND mapping added will be in the working set for subsequent submissions on that VM and will not be in the working set for currently running batches (which would require additional TLB flushes, which is not supported).

Execbuf ioctl in VM_BIND mode

A VM in VM_BIND mode will not support older execbuf mode of binding. The execbuf ioctl handling in VM_BIND mode differs significantly from the older execbuf2 ioctl (See *struct drm_i915_gem_execbuffer2*). Hence, a new execbuf3 ioctl has been added to support VM_BIND mode. (See *struct drm_i915_gem_execbuffer3*). The execbuf3 ioctl will not accept any execlist. Hence, no support for implicit sync. It is expected that the below work will be able to support requirements of object dependency setting in all use cases:

“dma-buf: Add an API for exporting sync files” (<https://lwn.net/Articles/859290/>)

The new execbuf3 ioctl only works in VM_BIND mode and the VM_BIND mode only works with execbuf3 ioctl for submission. All BOs mapped on that VM (through VM_BIND call) at the time of execbuf3 call are deemed required for that submission.

The execbuf3 ioctl directly specifies the batch addresses instead of as object handles as in execbuf2 ioctl. The execbuf3 ioctl will also not support many of the older features like in/out/submit fences, fence array, default gem context and many more (See *struct drm_i915_gem_execbuffer3*).

In VM_BIND mode, VA allocation is completely managed by the user instead of the i915 driver. Hence all VA assignment, eviction are not applicable in VM_BIND mode. Also, for determining object activeness, VM_BIND mode will not be using the i915_vma active reference tracking. It will instead use dma-resv object for that (See *VM_BIND dma_resv usage*).

So, a lot of existing code supporting execbuf2 ioctl, like relocations, VA evictions, vma lookup table, implicit sync, vma active reference tracking etc., are not applicable for execbuf3 ioctl. Hence, all execbuf3 specific handling should be in a separate file and only functionalities common to these ioctls can be the shared code where possible.

VM_PRIVATE objects

By default, BOs can be mapped on multiple VMs and can also be dma-buf exported. Hence these BOs are referred to as Shared BOs. During each execbuf submission, the request fence must be added to the dma-resv fence list of all shared BOs mapped on the VM.

VM_BIND feature introduces an optimization where user can create BO which is private to a specified VM via I915_GEM_CREATE_EXT_VM_PRIVATE flag during BO creation. Unlike Shared BOs, these VM private BOs can only be mapped on the VM they are private to and can't be dma-buf exported. All private BOs of a VM share the dma-resv object. Hence during each execbuf submission, they need only one dma-resv fence list updated. Thus, the fast path (where required mappings are already bound) submission latency is O(1) w.r.t the number of VM private BOs.

VM_BIND locking hierarchy

The locking design here supports the older (execlist based) execbuf mode, the newer VM_BIND mode, the VM_BIND mode with GPU page faults and possible future system allocator support (See [Shared Virtual Memory \(SVM\) support](#)). The older execbuf mode and the newer VM_BIND mode without page faults manages residency of backing storage using dma_fence. The VM_BIND mode with page faults and the system allocator support do not use any dma_fence at all.

VM_BIND locking order is as below.

- 1) Lock-A: A vm_bind mutex will protect vm_bind lists. This lock is taken in vm_bind/vm_unbind ioctl calls, in the execbuf path and while releasing the mapping.

In future, when GPU page faults are supported, we can potentially use a rwsem instead, so that multiple page fault handlers can take the read side lock to lookup the mapping and hence can run in parallel. The older execbuf mode of binding do not need this lock.

- 2) Lock-B: The object's dma-resv lock will protect i915_vma state and needs to be held while binding/unbinding a vma in the async worker and while updating dma-resv fence list of an object. Note that private BOs of a VM will all share a dma-resv object.

The future system allocator support will use the HMM prescribed locking instead.

- 3) Lock-C: Spinlock/s to protect some of the VM's lists like the list of invalidated vmas (due to eviction and userptr invalidation) etc.

When GPU page faults are supported, the execbuf path do not take any of these locks. There we will simply smash the new batch buffer address into the ring and then tell the scheduler run that. The lock taking only happens from the page fault handler, where we take lock-A in read mode, whichever lock-B we need to find the backing storage (dma_resv lock for gem objects, and hmm/core mm for system allocator) and some additional locks (lock-D) for taking care of page table races. Page fault mode should not need to ever manipulate the vm lists, so won't ever need lock-C.

VM_BIND LRU handling

We need to ensure VM_BIND mapped objects are properly LRU tagged to avoid performance degradation. We will also need support for bulk LRU movement of VM_BIND objects to avoid additional latencies in execbuf path.

The page table pages are similar to VM_BIND mapped objects (See [Evictable page table allocations](#)) and are maintained per VM and needs to be pinned in memory when VM is made active (ie., upon an execbuf call with that VM). So, bulk LRU movement of page table pages is also needed.

VM_BIND dma_resv usage

Fences needs to be added to all VM_BIND mapped objects. During each execbuf submission, they are added with DMA_RESV_USAGE_BOOKKEEP usage to prevent over sync (See enum dma_resv_usage). One can override it with either DMA_RESV_USAGE_READ or DMA_RESV_USAGE_WRITE usage during explicit object dependency setting.

Note that DRM_I915_GEM_WAIT and DRM_I915_GEM_BUSY ioctls do not check for DMA_RESV_USAGE_BOOKKEEP usage and hence should not be used for end of batch check. Instead, the execbuf3 out fence should be used for end of batch check (See *struct drm_i915_gem_execbuffer3*).

Also, in VM_BIND mode, use dma-resv apis for determining object activeness (See dma_resv_test_signaled() and dma_resv_wait_timeout()) and do not use the older i915_vma active reference tracking which is deprecated. This should be easier to get it working with the current TTM backend.

Mesa use case

VM_BIND can potentially reduce the CPU overhead in Mesa (both Vulkan and Iris), hence improving performance of CPU-bound applications. It also allows us to implement Vulkan's Sparse Resources. With increasing GPU hardware performance, reducing CPU overhead becomes more impactful.

16.4.2 Other VM_BIND use cases

Long running Compute contexts

Usage of dma-fence expects that they complete in reasonable amount of time. Compute on the other hand can be long running. Hence it is appropriate for compute to use user/memory fence (See *User/Memory Fence*) and dma-fence usage must be limited to in-kernel consumption only.

Where GPU page faults are not available, kernel driver upon buffer invalidation will initiate a suspend (preemption) of long running context, finish the invalidation, revalidate the BO and then resume the compute context. This is done by having a per-context preempt fence which is enabled when someone tries to wait on it and triggers the context preemption.

User/Memory Fence

User/Memory fence is a <address, value> pair. To signal the user fence, the specified value will be written at the specified virtual address and wakeup the waiting process. User fence can be signaled either by the GPU or kernel async worker (like upon bind completion). User can wait on a user fence with a new user fence wait ioctl.

Here is some prior work on this: <https://patchwork.freedesktop.org/patch/349417/>

Low Latency Submission

Allows compute UMD to directly submit GPU jobs instead of through `execbuf ioctl`. This is made possible by `VM_BIND` is not being synchronized against `execbuf`. `VM_BIND` allows bind/unbind of mappings required for the directly submitted jobs.

Debugger

With debug event interface user space process (debugger) is able to keep track of and act upon resources created by another process (debugged) and attached to GPU via `vm_bind` interface.

GPU page faults

GPU page faults when supported (in future), will only be supported in the `VM_BIND` mode. While both the older `execbuf` mode and the newer `VM_BIND` mode of binding will require using `dma-fence` to ensure residency, the GPU page faults mode when supported, will not use any `dma-fence` as residency is purely managed by installing and removing/invalidating page table entries.

Page level hints settings

`VM_BIND` allows any hints setting per mapping instead of per BO. Possible hints include placement and atomicity. Sub-BO level placement hint will be even more relevant with upcoming GPU on-demand page fault support.

Page level Cache/CLOS settings

`VM_BIND` allows cache/CLOS settings per mapping instead of per BO.

Evictable page table allocations

Make pagetable allocations evictable and manage them similar to `VM_BIND` mapped objects. Page table pages are similar to persistent mappings of a VM (difference here are that the page table pages will not have an `i915_vma` structure and after swapping pages back in, parent page link needs to be updated).

Shared Virtual Memory (SVM) support

`VM_BIND` interface can be used to map system memory directly (without gem BO abstraction) using the HMM interface. SVM is only supported with GPU page faults enabled.

16.4.3 VM_BIND UAPI

I915_PARAM_VM_BIND_VERSION

VM_BIND feature version supported. See `typedef drm_i915_getparam_t` param.

Specifies the VM_BIND feature version supported. The following versions of VM_BIND have been defined:

0: No VM_BIND support.

1: In VM_UNBIND calls, the UMD must specify the exact mappings created
previously with VM_BIND, the ioctl will not support unbinding multiple mappings or splitting them. Similarly, VM_BIND calls will not replace any existing mappings.

2: The restrictions on unbinding partial or multiple mappings is
lifted, Similarly, binding will replace any mappings in the given range.

See `struct drm_i915_gem_vm_bind` and `struct drm_i915_gem_vm_unbind`.

I915_VM_CREATE_FLAGS_USE_VM_BIND

Flag to opt-in for VM_BIND mode of binding during VM creation. See `struct drm_i915_gem_vm_control` flags.

The older `execbuf2` ioctl will not support VM_BIND mode of operation. For VM_BIND mode, we have new `execbuf3` ioctl which will not accept any `execlist` (See `struct drm_i915_gem_execbuffer3` for more details).

struct `drm_i915_gem_timeline_fence`

An input or output timeline fence.

Definition:

```
struct drm_i915_gem_timeline_fence {
    __u32 handle;
    __u32 flags;
#define I915_TIMELINE_FENCE_WAIT          (1 << 0);
#define I915_TIMELINE_FENCE_SIGNAL      (1 << 1);
#define __I915_TIMELINE_FENCE_UNKNOWN_FLAGS (- (I915_TIMELINE_FENCE_SIGNAL << 1));
    __u64 value;
};
```

Members

handle

User's handle for a `drm_syncobj` to wait on or signal.

flags

Supported flags are:

`I915_TIMELINE_FENCE_WAIT`: Wait for the input fence before the operation.

`I915_TIMELINE_FENCE_SIGNAL`: Return operation completion fence as output.

value

A point in the timeline. Value must be 0 for a binary `drm_syncobj`. A Value of 0 for a timeline `drm_syncobj` is invalid as it turns a `drm_syncobj` into a binary one.

Description

The operation will wait for input fence to signal.

The returned output fence will be signaled after the completion of the operation.

struct **drm_i915_gem_vm_bind**

VA to object mapping to bind.

Definition:

```

struct drm_i915_gem_vm_bind {
    __u32 vm_id;
    __u32 handle;
    __u64 start;
    __u64 offset;
    __u64 length;
    __u64 flags;
#define I915_GEM_VM_BIND_CAPTURE      (1 << 0);
    struct drm_i915_gem_timeline_fence fence;
    __u64 extensions;
};

```

Members

vm_id

VM (address space) id to bind

handle

Object handle

start

Virtual Address start to bind

offset

Offset in object to bind

length

Length of mapping to bind

flags

Supported flags are:

I915_GEM_VM_BIND_CAPTURE: Capture this mapping in the dump upon GPU error.

Note that **fence** carries its own flags.

fence

Timeline fence for bind completion signaling.

Timeline fence is of format *struct drm_i915_gem_timeline_fence*.

It is an out fence, hence using I915_TIMELINE_FENCE_WAIT flag is invalid, and an error will be returned.

If I915_TIMELINE_FENCE_SIGNAL flag is not set, then out fence is not requested and binding is completed synchronously.

extensions

Zero-terminated chain of extensions.

For future extensions. See *struct i915_user_extension*.

Description

This structure is passed to VM_BIND ioctl and specifies the mapping of GPU virtual address (VA) range to the section of an object that should be bound in the device page table of the specified address space (VM). The VA range specified must be unique (ie., not currently bound) and can be mapped to whole object or a section of the object (partial binding). Multiple VA mappings can be created to the same section of the object (aliasing).

The **start**, **offset** and **length** must be 4K page aligned. However the DG2 and XEHPSDV has 64K page size for device local memory and has compact page table. On those platforms, for binding device local-memory objects, the **start**, **offset** and **length** must be 64K aligned. Also, UMDs should not mix the local memory 64K page and the system memory 4K page bindings in the same 2M range.

Error code -EINVAL will be returned if **start**, **offset** and **length** are not properly aligned. In version 1 (See I915_PARAM_VM_BIND_VERSION), error code -ENOSPC will be returned if the VA range specified can't be reserved.

VM_BIND/UNBIND ioctl calls executed on different CPU threads concurrently are not ordered. Furthermore, parts of the VM_BIND operation can be done asynchronously, if valid **fence** is specified.

struct **drm_i915_gem_vm_unbind**

VA to object mapping to unbind.

Definition:

```
struct drm_i915_gem_vm_unbind {
    __u32 vm_id;
    __u32 rsvd;
    __u64 start;
    __u64 length;
    __u64 flags;
    struct drm_i915_gem_timeline_fence fence;
    __u64 extensions;
};
```

Members

vm_id

VM (address space) id to bind

rsvd

Reserved, MBZ

start

Virtual Address start to unbind

length

Length of mapping to unbind

flags

Currently reserved, MBZ.

Note that **fence** carries its own flags.

fence

Timeline fence for unbind completion signaling.

Timeline fence is of format *struct drm_i915_gem_timeline_fence*.

It is an out fence, hence using I915_TIMELINE_FENCE_WAIT flag is invalid, and an error will be returned.

If I915_TIMELINE_FENCE_SIGNAL flag is not set, then out fence is not requested and unbinding is completed synchronously.

extensions

Zero-terminated chain of extensions.

For future extensions. See *struct i915_user_extension*.

Description

This structure is passed to VM_UNBIND ioctl and specifies the GPU virtual address (VA) range that should be unbound from the device page table of the specified address space (VM). VM_UNBIND will force unbind the specified range from device page table without waiting for any GPU job to complete. It is UMDs responsibility to ensure the mapping is no longer in use before calling VM_UNBIND.

If the specified mapping is not found, the ioctl will simply return without any error.

VM_BIND/UNBIND ioctl calls executed on different CPU threads concurrently are not ordered. Furthermore, parts of the VM_UNBIND operation can be done asynchronously, if valid **fence** is specified.

struct drm_i915_gem_execbuffer3

Structure for DRM_I915_GEM_EXECBUFFER3 ioctl.

Definition:

```
struct drm_i915_gem_execbuffer3 {
    __u32 ctx_id;
    __u32 engine_idx;
    __u64 batch_address;
    __u64 flags;
    __u32 rsvd1;
    __u32 fence_count;
    __u64 timeline_fences;
    __u64 rsvd2;
    __u64 extensions;
};
```

Members**ctx_id**

Context id

Only contexts with user engine map are allowed.

engine_idx

Engine index

An index in the user engine map of the context specified by **ctx_id**.

batch_address

Batch gpu virtual address/es.

For normal submission, it is the gpu virtual address of the batch buffer. For parallel submission, it is a pointer to an array of batch buffer gpu virtual addresses with array size equal to the number of (parallel) engines involved in that submission (See [*struct i915_context_engines_parallel_submit*](#)).

flags

Currently reserved, MBZ

rsvd1

Reserved, MBZ

fence_count

Number of fences in **timeline_fences** array.

timeline_fences

Pointer to an array of timeline fences.

Timeline fences are of format [*struct drm_i915_gem_timeline_fence*](#).

rsvd2

Reserved, MBZ

extensions

Zero-terminated chain of extensions.

For future extensions. See [*struct i915_user_extension*](#).

Description

DRM_I915_GEM_EXECBUFFER3 ioctl only works in VM_BIND mode and VM_BIND mode only works with this ioctl for submission. See I915_VM_CREATE_FLAGS_USE_VM_BIND.

struct `drm_i915_gem_create_ext_vm_private`

Extension to make the object private to the specified VM.

Definition:

```
struct drm_i915_gem_create_ext_vm_private {
#define I915_GEM_CREATE_EXT_VM_PRIVATE      2;
    struct i915_user_extension base;
    __u32 vm_id;
};
```

Members

base

Extension link. See [*struct i915_user_extension*](#).

vm_id

Id of the VM to which the object is private

Description

See [*struct drm_i915_gem_create_ext*](#).

16.5 Xe - Merge Acceptance Plan

Xe is a new driver for Intel GPUs that supports both integrated and discrete platforms starting with Tiger Lake (first Intel Xe Architecture).

This document aims to establish a merge plan for the Xe, by writing down clear pre-merge goals, in order to avoid unnecessary delays.

16.5.1 Xe - Overview

The main motivation of Xe is to have a fresh base to work from that is unencumbered by older platforms, whilst also taking the opportunity to rearchitect our driver to increase sharing across the drm subsystem, both leveraging and allowing us to contribute more towards other shared components like TTM and drm/scheduler.

This is also an opportunity to start from the beginning with a clean uAPI that is extensible by design and already aligned with the modern userspace needs. For this reason, the memory model is solely based on GPU Virtual Address space bind/unbind ('VM_BIND') of GEM buffer objects (BOs) and execution only supporting explicit synchronization. With persistent mapping across the execution, the userspace does not need to provide a list of all required mappings during each submission.

The new driver leverages a lot from i915. As for display, the intent is to share the display code with the i915 driver so that there is maximum reuse there.

As for the power management area, the goal is to have a much-simplified support for the system suspend states (S-states), PCI device suspend states (D-states), GPU/Render suspend states (R-states) and frequency management. It should leverage as much as possible all the existent PCI-subsystem infrastructure (pm and runtime_pm) and underlying firmware components such PCODE and GuC for the power states and frequency decisions.

Repository:

<https://gitlab.freedesktop.org/drm/xe/kernel> (branch drm-xe-next)

16.5.2 Xe - Platforms

Currently, Xe is already functional and has experimental support for multiple platforms starting from Tiger Lake, with initial support in userspace implemented in Mesa (for Iris and Anv, our OpenGL and Vulkan drivers), as well as in NEO (for OpenCL and Level0).

During a transition period, platforms will be supported by both Xe and i915. However, the force_probe mechanism existent in both drivers will allow only one official and by-default probe at a given time.

For instance, in order to probe a DG2 which PCI ID is 0x5690 by Xe instead of i915, the following set of parameters need to be used:

```
` i915.force_probe=!5690 xe.force_probe=5690 `
```

In both drivers, the 'require_force_probe' protection forces the user to use the force_probe parameter while the driver is under development. This protection is only removed when the support for the platform and the uAPI are stable. Stability which needs to be demonstrated by CI results.

In order to avoid user space regressions, i915 will continue to support all the current platforms that are already out of this protection. Xe support will be forever experimental and dependent on the usage of `force_probe` for these platforms.

When the time comes for Xe, the protection will be lifted on Xe and kept in i915.

Xe driver will be protected with both STAGING Kconfig and `force_probe`. Changes in the uAPI are expected while the driver is behind these protections. STAGING will be removed when the driver uAPI gets to a mature state where we can guarantee the ‘no regression’ rule. Then `force_probe` will be lifted only for future platforms that will be productized with Xe driver, but not with i915.

16.5.3 Xe - Pre-Merge Goals

Drm_scheduler

Xe primarily uses Firmware based scheduling (GuC FW). However, it will use `drm_scheduler` as the scheduler ‘frontend’ for userspace submission in order to resolve syncobj and dma-buf implicit sync dependencies. However, `drm_scheduler` is not yet prepared to handle the 1-to-1 relationship between `drm_gpu_scheduler` and `drm_sched_entity`.

Deeper changes to `drm_scheduler` should *not* be required to get Xe accepted, but some consensus needs to be reached between Xe and other community drivers that could also benefit from this work, for coupling FW based/assisted submission such as the ARM’s new Mali GPU driver, and others.

As a key measurable result, the patch series introducing Xe itself shall not depend on any other patch touching `drm_scheduler` itself that was not yet merged through `drm-misc`. This, by itself, already includes the reach of an agreement for uniform 1 to 1 relationship implementation / usage across drivers.

GPU VA

Two main goals of Xe are meeting together here:

- 1) Have an uAPI that aligns with modern UMD needs.
- 2) Early upstream engagement.

RedHat engineers working on Nouveau proposed a new DRM feature to handle keeping track of GPU virtual address mappings. This is still not merged upstream, but this aligns very well with our goals and with our `VM_BIND`. The engagement with upstream and the port of Xe towards GPUVA is already ongoing.

As a key measurable result, Xe needs to be aligned with the GPU VA and working in our tree. Missing Nouveau patches should *not* block Xe and any needed GPUVA related patch should be independent and present on `dri-devel` or acked by maintainers to go along with the first Xe pull request towards `drm-next`.

DRM_VM_BIND

Nouveau, and Xe are all implementing 'VM_BIND' and new 'Exec' uAPIs in order to fulfill the needs of the modern uAPI. Xe merge should *not* be blocked on the development of a common new drm_infrastructure. However, the Xe team needs to engage with the community to explore the options of a common API.

As a key measurable result, the DRM_VM_BIND needs to be documented in this file below, or this entire block deleted if the consensus is for independent drivers vm_bind ioctls.

Although having a common DRM level IOCTL for VM_BIND is not a requirement to get Xe merged, it is mandatory to enforce the overall locking scheme for all major structs and list (so vm and vma). So, a consensus is needed, and possibly some common helpers. If helpers are needed, they should be also documented in this document.

ASYNC VM_BIND

Although having a common DRM level IOCTL for VM_BIND is not a requirement to get Xe merged, it is mandatory to have a consensus with other drivers and Mesa. It needs to be clear how to handle async VM_BIND and interactions with userspace memory fences. Ideally with helper support so people don't get it wrong in all possible ways.

As a key measurable result, the benefits of ASYNC VM_BIND and a discussion of various flavors, error handling and a sample API should be documented here or in a separate document pointed to by this document.

Userptr integration and vm_bind

Different drivers implement different ways of dealing with execution of userptr. With multiple drivers currently introducing support to VM_BIND, the goal is to aim for a DRM consensus on what's the best way to have that support. To some extent this is already getting addressed itself with the GPUVA where likely the userptr will be a GPUVA with a NULL GEM call VM bind directly on the userptr. However, there are more aspects around the rules for that and the usage of mmu_notifiers, locking and other aspects.

This task here has the goal of introducing a documentation of the basic rules.

The documentation *needs* to first live in this document (API session below) and then moved to another more specific document or at Xe level or at DRM level.

Documentation should include:

- The userptr part of the VM_BIND api.
- Locking, including the page-faulting case.
- O(1) complexity under VM_BIND.

Some parts of userptr like mmu_notifiers should become GPUVA or DRM helpers when the second driver supporting VM_BIND+userptr appears. Details to be defined when the time comes.

Long running compute: minimal data structure/scaffolding

The generic scheduler code needs to include the handling of endless compute contexts, with the minimal scaffolding for preempt-ctx fences (probably on the `drm_sched_entity`) and making sure `drm_scheduler` can cope with the lack of job completion fence.

The goal is to achieve a consensus ahead of Xe initial pull-request, ideally with this minimal `drm/scheduler` work, if needed, merged to `drm-misc` in a way that any `drm` driver, including Xe, could re-use and add their own individual needs on top in a next stage. However, this should not block the initial merge.

This is a non-blocker item since the driver without the support for the long running compute enabled is not a showstopper.

Display integration with i915

In order to share the display code with the `i915` driver so that there is maximum reuse, the `i915/display/` code is built twice, once for `i915.ko` and then for `xe.ko`. Currently, the `i915/display` code in Xe tree is polluted with many ‘`ifdefs`’ depending on the build target. The goal is to refactor both Xe and `i915/display` code simultaneously in order to get a clean result before they land upstream, so that display can already be part of the initial pull request towards `drm-next`.

However, display code should not gate the acceptance of Xe in upstream. Xe patches will be refactored in a way that display code can be removed, if needed, from the first pull request of Xe towards `drm-next`. The expectation is that when both drivers are part of the `drm-tip`, the introduction of cleaner patches will be easier and speed up.

Drm_exec

Helper to make `dma_resv` locking for a big number of buffers is getting removed in the `drm_exec` series proposed in <https://patchwork.freedesktop.org/patch/524376/>. If that happens, Xe needs to change and incorporate the changes in the driver. The goal is to engage with the Community to understand if the best approach is to move that to the drivers that are using it or if we should keep the helpers in place waiting for Xe to get merged.

This item ties into the GPUVA, `VM_BIND`, and even long-running compute support.

As a key measurable result, we need to have a community consensus documented in this document and the Xe driver prepared for the changes, if necessary.

Dev_coredump

Xe needs to align with other drivers on the way that the error states are dumped, avoiding a Xe only `error_state` solution. The goal is to use `devcoredump` infrastructure to report error states, since it produces a standardized way by exposing a virtual and temporary `/sys/class/devcoredump` device.

As the key measurable result, Xe driver needs to provide GPU snapshots captured at hang time through `devcoredump`, but without depending on any core modification of `devcoredump` infrastructure itself.

Later, when we are in-tree, the goal is to collaborate with `devcoredump` infrastructure with overall possible improvements, like multiple file support for better organization of the dumps,

snapshot support, dmesg extra print, and whatever may make sense and help the overall infrastructure.

16.5.4 Xe - uAPI high level overview

...Warning: To be done in follow up patches after/when/where the main consensus in various items are individually reached.

Symbols

__drm_atomic_get_current_plane_state (C function), 238
__drm_atomic_helper_bridge_duplicate_state (C function), 509
__drm_atomic_helper_bridge_reset (C function), 509
__drm_atomic_helper_connector_destroy_state (C function), 508
__drm_atomic_helper_connector_duplicate_state (C function), 507
__drm_atomic_helper_connector_reset (C function), 506
__drm_atomic_helper_connector_state_reset (C function), 506
__drm_atomic_helper_crtc_destroy_state (C function), 504
__drm_atomic_helper_crtc_duplicate_state (C function), 503
__drm_atomic_helper_crtc_reset (C function), 502
__drm_atomic_helper_crtc_state_reset (C function), 502
__drm_atomic_helper_plane_destroy_state (C function), 505
__drm_atomic_helper_plane_duplicate_state (C function), 505
__drm_atomic_helper_plane_reset (C function), 504
__drm_atomic_helper_plane_state_reset (C function), 504
__drm_atomic_helper_private_obj_duplicate_state (C function), 508
__drm_atomic_state_free (C function), 247
__drm_ctm_to_dc_matrix (C function), 915
__drm_dp_mst_state_iter_get (C function), 635
__drm_gem_destroy_shadow_plane_state (C function), 514
__drm_gem_duplicate_shadow_plane_state (C function), 514
__drm_gem_reset_shadow_plane (C function), 515
__drm_i915_gem_create_ext (C struct), 1199
drm_i915_memory_region_info (C struct), 1201
__drm_lut_to_dc_gamma (C function), 915
__extract_blob_lut (C function), 915
hostlx_client_init (C function), 1101
__hostlx_client_register (C function), 1102
__i915_gem_object_make_purgeable (C function), 1023
__i915_gem_object_make_shrinkable (C function), 1023
__intel_runtime_pm_get_if_active (C function), 964
__intel_wait_for_register (C function), 968
__intel_wait_for_register_fw (C function), 968
__is_lut_linear (C function), 915
__set_input_tf (C function), 916
__set_legacy_tf (C function), 916
__set_output_tf (C function), 916

A

ABM, 939
active_cu_number, 961
amd_ip_block_type (C enum), 896
amd_ip_funcs (C struct), 897
amdgpu_bo_add_to_shadow_list (C function), 860
amdgpu_bo_create (C function), 859
amdgpu_bo_create_kernel (C function), 857
amdgpu_bo_create_kernel_at (C function), 858
amdgpu_bo_create_reserved (C function), 856
amdgpu_bo_create_user (C function), 859
amdgpu_bo_create_vm (C function), 859

`amdgpu_bo_fault_reserve_notify` (C function), 865

`amdgpu_bo_fence` (C function), 865

`amdgpu_bo_fini` (C function), 863

`amdgpu_bo_free_kernel` (C function), 858

`amdgpu_bo_get_metadata` (C function), 864

`amdgpu_bo_get_preferred_domain` (C function), 867

`amdgpu_bo_get_tiling_flags` (C function), 863

`amdgpu_bo_gpu_offset` (C function), 866

`amdgpu_bo_gpu_offset_no_check` (C function), 867

`amdgpu_bo_init` (C function), 863

`amdgpu_bo_is_amdgpu_bo` (C function), 856

`amdgpu_bo_kmap` (C function), 860

`amdgpu_bo_kptr` (C function), 860

`amdgpu_bo_kunmap` (C function), 861

`amdgpu_bo_move_notify` (C function), 864

`amdgpu_bo_pin` (C function), 862

`amdgpu_bo_pin_restricted` (C function), 861

`amdgpu_bo_placement_from_domain` (C function), 856

`amdgpu_bo_print_info` (C function), 867

`amdgpu_bo_ref` (C function), 861

`amdgpu_bo_release_notify` (C function), 865

`amdgpu_bo_restore_shadow` (C function), 860

`amdgpu_bo_set_metadata` (C function), 864

`amdgpu_bo_set_tiling_flags` (C function), 863

`amdgpu_bo_sync_wait` (C function), 866

`amdgpu_bo_sync_wait_resv` (C function), 865

`amdgpu_bo_unpin` (C function), 862

`amdgpu_bo_unref` (C function), 861

`amdgpu_debugfs_vm_bo_info` (C function), 890

`amdgpu_display_manager` (C struct), 903

`amdgpu_dm_atomic_check` (C function), 913

`amdgpu_dm_atomic_commit_tail` (C function), 913

`amdgpu_dm_backlight_caps` (C struct), 901

`amdgpu_dm_hpd_fini` (C function), 912

`amdgpu_dm_hpd_init` (C function), 912

`amdgpu_dm_init_color_mod` (C function), 914

`amdgpu_dm_irq_fini` (C function), 911

`amdgpu_dm_irq_handler` (C function), 912

`amdgpu_dm_irq_handler_data` (C struct), 909

`amdgpu_dm_irq_init` (C function), 911

`amdgpu_dm_irq_register_interrupt` (C function), 910

`amdgpu_dm_irq_unregister_interrupt` (C function), 911

`amdgpu_dm_update_crtc_color_mgmt` (C function), 917

`amdgpu_dm_update_plane_color_mgmt` (C function), 917

`amdgpu_dm_verify_lut_sizes` (C function), 917

`amdgpu_dma_buf_attach` (C function), 868

`amdgpu_dma_buf_begin_cpu_access` (C function), 869

`amdgpu_dma_buf_create_obj` (C function), 870

`amdgpu_dma_buf_detach` (C function), 868

`amdgpu_dma_buf_map` (C function), 868

`amdgpu_dma_buf_move_notify` (C function), 870

`amdgpu_dma_buf_pin` (C function), 868

`amdgpu_dma_buf_unmap` (C function), 869

`amdgpu_dma_buf_unpin` (C function), 868

`amdgpu_dmabuf_is_xgmi_accessible` (C function), 871

`amdgpu_drv_delayed_reset_work_handler` (C function), 853

`amdgpu_gem_prime_export` (C function), 869

`amdgpu_gem_prime_import` (C function), 870

`amdgpu_hdmi_vsdb_info` (C struct), 907

`amdgpu_hmm_invalidate_gfx` (C function), 871

`amdgpu_hmm_invalidate_hsa` (C function), 871

`amdgpu_hmm_register` (C function), 872

`amdgpu_hmm_unregister` (C function), 872

`amdgpu_irq_add_domain` (C function), 895

`amdgpu_irq_add_id` (C function), 892

`amdgpu_irq_create_mapping` (C function), 896

`amdgpu_irq_delegate` (C function), 893

`amdgpu_irq_disable_all` (C function), 890

`amdgpu_irq_dispatch` (C function), 893

`amdgpu_irq_enabled` (C function), 894

`amdgpu_irq_fini_sw` (C function), 892

`amdgpu_irq_get` (C function), 894

`amdgpu_irq_gpu_reset_resume_helper` (C function), 893

`amdgpu_irq_handle_ih1` (C function), 891

`amdgpu_irq_handle_ih2` (C function), 891

`amdgpu_irq_handle_ih_soft` (C function), 891

`amdgpu_irq_handler` (C function), 890

`amdgpu_irq_init` (C function), 892

`amdgpu_irq_put` (C function), 894

- amdgpu_irq_remove_domain (C function), 895
- amdgpu_irq_update (C function), 893
- amdgpu_irqdomain_map (C function), 895
- amdgpu_msi_ok (C function), 891
- amdgpu_prt_cb (C struct), 873
- amdgpu_vm_add_prt_cb (C function), 881
- amdgpu_vm_adjust_size (C function), 886
- amdgpu_vm_bo_add (C function), 882
- amdgpu_vm_bo_base_init (C function), 875
- amdgpu_vm_bo_clear_mappings (C function), 884
- amdgpu_vm_bo_del (C function), 885
- amdgpu_vm_bo_done (C function), 875
- amdgpu_vm_bo_evicted (C function), 874
- amdgpu_vm_bo_find (C function), 878
- amdgpu_vm_bo_idle (C function), 874
- amdgpu_vm_bo_insert_map (C function), 883
- amdgpu_vm_bo_invalidate (C function), 886
- amdgpu_vm_bo_invalidated (C function), 874
- amdgpu_vm_bo_lookup_mapping (C function), 885
- amdgpu_vm_bo_map (C function), 883
- amdgpu_vm_bo_moved (C function), 874
- amdgpu_vm_bo_relocated (C function), 874
- amdgpu_vm_bo_replace_map (C function), 883
- amdgpu_vm_bo_reset_state_machine (C function), 875
- amdgpu_vm_bo_trace_cs (C function), 885
- amdgpu_vm_bo_unmap (C function), 884
- amdgpu_vm_bo_update (C function), 880
- amdgpu_vm_check_compute_bug (C function), 877
- amdgpu_vm_clear_freed (C function), 882
- amdgpu_vm_evictable (C function), 886
- amdgpu_vm_fini (C function), 888
- amdgpu_vm_flush (C function), 877
- amdgpu_vm_free_mapping (C function), 881
- amdgpu_vm_generation (C function), 876
- amdgpu_vm_get_block_size (C function), 886
- amdgpu_vm_get_task_info (C function), 889
- amdgpu_vm_handle_fault (C function), 889
- amdgpu_vm_handle_moved (C function), 882
- amdgpu_vm_init (C function), 887
- amdgpu_vm_ioctl (C function), 889
- amdgpu_vm_lock_pd (C function), 875
- amdgpu_vm_make_compute (C function), 887
- amdgpu_vm_manager_fini (C function), 888
- amdgpu_vm_manager_init (C function), 888
- amdgpu_vm_map_gart (C function), 878
- amdgpu_vm_move_to_lru_tail (C function), 876
- amdgpu_vm_need_pipeline_sync (C function), 877
- amdgpu_vm_prt_cb (C function), 881
- amdgpu_vm_prt_fini (C function), 881
- amdgpu_vm_prt_get (C function), 880
- amdgpu_vm_prt_put (C function), 880
- amdgpu_vm_ready (C function), 877
- amdgpu_vm_release_compute (C function), 888
- amdgpu_vm_set_pasid (C function), 873
- amdgpu_vm_set_task_info (C function), 889
- amdgpu_vm_tlb_seq_cb (C function), 879
- amdgpu_vm_tlb_seq_struct (C struct), 873
- amdgpu_vm_update_pdes (C function), 878
- amdgpu_vm_update_prt_state (C function), 880
- amdgpu_vm_update_range (C function), 879
- amdgpu_vm_validate_pt_bos (C function), 876
- amdgpu_vm_wait_idle (C function), 887
- append_oa_sample (C function), 1086
- append_oa_status (C function), 1085
- apple_gmux_detect (C function), 1166
- apple_gmux_present (C function), 1166
- APU, 939
- ARB, 940
- ASIC, 939
- ASSR, 939
- AZ, 939
- B**
- backlight_device (C struct), 1147
- backlight_device_get_by_name (C function), 1149
- backlight_disable (C function), 1148
- backlight_enable (C function), 1148
- backlight_force_update (C function), 1149
- backlight_get_brightness (C function), 1148
- backlight_is_blank (C function), 1148
- backlight_notification (C enum), 1144
- backlight_ops (C struct), 1144
- backlight_properties (C struct), 1145
- backlight_register_notifier (C function), 1150
- backlight_type (C enum), 1144
- backlight_unregister_notifier (C function), 1150
- backlight_update_reason (C enum), 1143
- backlight_update_status (C function), 1148
- bdb_header (C struct), 999

bl_get_data (C function), 1149

BPC, 939

BPP, 939

C

Clocks, 939

CP, 961

CPLIB, 961

CRC, 939

CRTC, 939

CU, 961

CVT, 939

D

DAL, 939

dal_allocation (C struct), 902

DC (Hardware), 940

DC (Software), 940

dc_color_caps (C struct), 920

dc_validation_set (C struct), 921

DCC, 940

DCCG, 940

DCE, 940

DCHUB, 940

DCN, 940

DDC, 940

DEFINE_DRM_GEM_DMA_FOPS (C macro), 94

DEFINE_DRM_GEM_FOPS (C macro), 80

detect_bit_6_swizzle (C function), 1042

devm_aperture_acquire_from_firmware (C function), 8

devm_backlight_device_register (C function), 1150

devm_backlight_device_unregister (C function), 1151

devm_drm_bridge_add (C function), 566

devm_drm_dev_alloc (C macro), 21

devm_drm_of_get_bridge (C function), 576

devm_drm_panel_add_follower (C function), 584

devm_drm_panel_bridge_add (C function), 575

devm_drm_panel_bridge_add_typed (C function), 575

devm_mipi_dsi_attach (C function), 669

devm_mipi_dsi_device_register_full (C function), 668

devm_of_find_backlight (C function), 1151

DFS, 961

dGPU, 940

DIO, 940

dm_compressor_info (C struct), 900

dm_crtc_high_irq (C function), 913

dm_hw_fini (C function), 909

dm_hw_init (C function), 908

dm_irq_work_func (C function), 910

dm_pflip_high_irq (C function), 912

DMCU, 940

DMCUB, 940

DMIF, 940

DML, 940

dmub_hpd_work (C struct), 900

dp_colorimetry (C enum), 589

dp_content_type (C enum), 591

dp_dynamic_range (C enum), 590

dp_pixelformat (C enum), 589

dp_sdp (C struct), 588

dp_sdp_header (C struct), 588

DPCD, 940

dppll_info (C struct), 1010

DPM(S), 940

DPP, 940

dpp_color_caps (C struct), 919

drm_add_edid_modes (C function), 708

drm_add_modes_noedid (C function), 708

drm_afbc_framebuffer (C struct), 286

drm_analog_tv_mode (C function), 329

drm_any_plane_has_format (C function), 312

drm_aperture_remove_conflicting_framebuffers (C function), 9

drm_aperture_remove_conflicting_pci_framebuffers (C function), 9

drm_aperture_remove_framebuffers (C function), 8

drm_atomic_add_affected_connectors (C function), 253

drm_atomic_add_affected_planes (C function), 254

drm_atomic_add_encoder_bridges (C function), 253

drm_atomic_bridge_chain_check (C function), 569

drm_atomic_bridge_chain_disable (C function), 568

drm_atomic_bridge_chain_enable (C function), 569

drm_atomic_bridge_chain_post_disable (C function), 568

drm_atomic_bridge_chain_pre_enable (C function), 569

drm_atomic_check_only (C function), 254

drm_atomic_commit (C function), 255

drm_atomic_crtc_effectively_active (C

function), 245

`drm_atomic_crtc_for_each_plane` (C macro), 481

`drm_atomic_crtc_needs_modeset` (C function), 244

`drm_atomic_crtc_state_for_each_plane` (C macro), 482

`drm_atomic_crtc_state_for_each_plane_state` (C macro), 482

`drm_atomic_for_each_plane_damage` (C macro), 320

`drm_atomic_get_bridge_state` (C function), 252

`drm_atomic_get_connector_state` (C function), 252

`drm_atomic_get_crtc_state` (C function), 248

`drm_atomic_get_existing_connector_state` (C function), 237

`drm_atomic_get_existing_crtc_state` (C function), 235

`drm_atomic_get_existing_plane_state` (C function), 236

`drm_atomic_get_mst_topology_state` (C function), 647

`drm_atomic_get_new_bridge_state` (C function), 253

`drm_atomic_get_new_connector_for_encoder` (C function), 250

`drm_atomic_get_new_connector_state` (C function), 237

`drm_atomic_get_new_crtc_for_encoder` (C function), 251

`drm_atomic_get_new_crtc_state` (C function), 235

`drm_atomic_get_new_mst_topology_state` (C function), 648

`drm_atomic_get_new_plane_state` (C function), 236

`drm_atomic_get_new_private_obj_state` (C function), 250

`drm_atomic_get_old_bridge_state` (C function), 252

`drm_atomic_get_old_connector_for_encoder` (C function), 250

`drm_atomic_get_old_connector_state` (C function), 237

`drm_atomic_get_old_crtc_for_encoder` (C function), 251

`drm_atomic_get_old_crtc_state` (C function), 235

`drm_atomic_get_old_mst_topology_state` (C function), 647

`drm_atomic_get_old_plane_state` (C function), 236

`drm_atomic_get_old_private_obj_state` (C function), 249

`drm_atomic_get_plane_state` (C function), 248

`drm_atomic_get_private_obj_state` (C function), 249

`drm_atomic_helper_async_check` (C function), 489

`drm_atomic_helper_async_commit` (C function), 490

`drm_atomic_helper_bridge_destroy_state` (C function), 509

`drm_atomic_helper_bridge_duplicate_state` (C function), 509

`drm_atomic_helper_bridge_propagate_bus_fmt` (C function), 501

`drm_atomic_helper_bridge_reset` (C function), 510

`drm_atomic_helper_calc_timestamping_constants` (C function), 487

`drm_atomic_helper_check` (C function), 486

`drm_atomic_helper_check_crtc_primary_plane` (C function), 485

`drm_atomic_helper_check_modeset` (C function), 483

`drm_atomic_helper_check_plane_damage` (C function), 318

`drm_atomic_helper_check_plane_state` (C function), 484

`drm_atomic_helper_check_planes` (C function), 485

`drm_atomic_helper_check_wb_encoder_state` (C function), 484

`drm_atomic_helper_cleanup_planes` (C function), 495

`drm_atomic_helper_commit` (C function), 490

`drm_atomic_helper_commit_cleanup_done` (C function), 492

`drm_atomic_helper_commit_duplicated_state` (C function), 499

`drm_atomic_helper_commit_hw_done` (C function), 492

`drm_atomic_helper_commit_modeset_disables` (C function), 487

`drm_atomic_helper_commit_modeset_enables` (C function), 487

`drm_atomic_helper_commit_planes` (C func-

tion), 493
drm_atomic_helper_commit_planes_on_crtc (C function), 494
drm_atomic_helper_commit_tail (C function), 489
drm_atomic_helper_commit_tail_rpm (C function), 489
drm_atomic_helper_connector_destroy_state (C function), 508
drm_atomic_helper_connector_duplicate_state (C function), 508
drm_atomic_helper_connector_reset (C function), 506
drm_atomic_helper_connector_tv_check (C function), 507
drm_atomic_helper_connector_tv_margins_reset (C function), 507
drm_atomic_helper_connector_tv_reset (C function), 507
drm_atomic_helper_crtc_destroy_state (C function), 504
drm_atomic_helper_crtc_duplicate_state (C function), 503
drm_atomic_helper_crtc_reset (C function), 503
drm_atomic_helper_damage_iter (C struct), 320
drm_atomic_helper_damage_iter_init (C function), 319
drm_atomic_helper_damage_iter_next (C function), 319
drm_atomic_helper_damage_merged (C function), 319
drm_atomic_helper_dirtyfb (C function), 318
drm_atomic_helper_disable_all (C function), 497
drm_atomic_helper_disable_plane (C function), 497
drm_atomic_helper_disable_planes_on_crtc (C function), 494
drm_atomic_helper_duplicate_state (C function), 498
drm_atomic_helper_fake_vblank (C function), 492
drm_atomic_helper_page_flip (C function), 500
drm_atomic_helper_page_flip_target (C function), 501
drm_atomic_helper_plane_destroy_state (C function), 506
drm_atomic_helper_plane_duplicate_state (C function), 505
drm_atomic_helper_plane_reset (C function), 505
drm_atomic_helper_prepare_planes (C function), 493
drm_atomic_helper_resume (C function), 500
drm_atomic_helper_set_config (C function), 497
drm_atomic_helper_setup_commit (C function), 491
drm_atomic_helper_shutdown (C function), 498
drm_atomic_helper_suspend (C function), 499
drm_atomic_helper_swap_state (C function), 495
drm_atomic_helper_unprepare_planes (C function), 493
drm_atomic_helper_update_legacy_modeset_state (C function), 486
drm_atomic_helper_update_plane (C function), 496
drm_atomic_helper_wait_for_dependencies (C function), 491
drm_atomic_helper_wait_for_fences (C function), 488
drm_atomic_helper_wait_for_flip_done (C function), 488
drm_atomic_helper_wait_for_vblanks (C function), 488
drm_atomic_nonblocking_commit (C function), 255
drm_atomic_normalize_zpos (C function), 316
drm_atomic_plane_disabling (C function), 483
drm_atomic_plane_enabling (C function), 482
drm_atomic_print_new_state (C function), 255
drm_atomic_private_obj_fini (C function), 249
drm_atomic_private_obj_init (C function), 248
drm_atomic_set_crtc_for_connector (C function), 258
drm_atomic_set_crtc_for_plane (C function), 257
drm_atomic_set_fb_for_plane (C function), 257

[drm_atomic_set_mode_for_crtc \(C function\), 256](#)
[drm_atomic_set_mode_prop_for_crtc \(C function\), 257](#)
[drm_atomic_state \(C struct\), 232](#)
[drm_atomic_state_alloc \(C function\), 247](#)
[drm_atomic_state_clear \(C function\), 247](#)
[drm_atomic_state_default_clear \(C function\), 247](#)
[drm_atomic_state_default_release \(C function\), 246](#)
[drm_atomic_state_get \(C function\), 234](#)
[drm_atomic_state_init \(C function\), 246](#)
[drm_atomic_state_put \(C function\), 235](#)
[drm_av_sync_delay \(C function\), 706](#)
[drm_bridge \(C struct\), 563](#)
[drm_bridge_add \(C function\), 565](#)
[drm_bridge_attach \(C function\), 566](#)
[drm_bridge_attach_flags \(C enum\), 554](#)
[drm_bridge_chain_get_first_bridge \(C function\), 565](#)
[drm_bridge_chain_mode_fixup \(C function\), 567](#)
[drm_bridge_chain_mode_set \(C function\), 567](#)
[drm_bridge_chain_mode_valid \(C function\), 567](#)
[drm_bridge_connector_init \(C function\), 573](#)
[drm_bridge_detect \(C function\), 570](#)
[drm_bridge_edid_read \(C function\), 570](#)
[drm_bridge_funcs \(C struct\), 554](#)
[drm_bridge_get_edid \(C function\), 571](#)
[drm_bridge_get_modes \(C function\), 570](#)
[drm_bridge_get_next_bridge \(C function\), 565](#)
[drm_bridge_get_prev_bridge \(C function\), 565](#)
[drm_bridge_hpd_disable \(C function\), 572](#)
[drm_bridge_hpd_enable \(C function\), 571](#)
[drm_bridge_hpd_notify \(C function\), 572](#)
[drm_bridge_is_panel \(C function\), 574](#)
[drm_bridge_ops \(C enum\), 563](#)
[drm_bridge_remove \(C function\), 566](#)
[drm_bridge_state \(C struct\), 246](#)
[drm_bridge_timings \(C struct\), 562](#)
[drm_buddy_alloc_blocks \(C function\), 172](#)
[drm_buddy_block_print \(C function\), 173](#)
[drm_buddy_block_trim \(C function\), 172](#)
[drm_buddy_fini \(C function\), 171](#)
[drm_buddy_free_block \(C function\), 172](#)
[drm_buddy_free_list \(C function\), 172](#)
[drm_buddy_init \(C function\), 171](#)
[drm_buddy_print \(C function\), 173](#)
[drm_bus_cfg \(C struct\), 245](#)
[drm_bus_flags \(C enum\), 349](#)
[drm_bus_flags_from_videomode \(C function\), 333](#)
[drm_calc_timestamping_constants \(C function\), 447](#)
[drm_can_sleep \(C function\), 50](#)
[DRM_CAP_ADDFB2_MODIFIERS \(C macro\), 777](#)
[DRM_CAP_ASYNC_PAGE_FLIP \(C macro\), 777](#)
[DRM_CAP_CRTC_IN_VBLANK_EVENT \(C macro\), 777](#)
[DRM_CAP_CURSOR_HEIGHT \(C macro\), 777](#)
[DRM_CAP_CURSOR_WIDTH \(C macro\), 777](#)
[DRM_CAP_DUMB_BUFFER \(C macro\), 775](#)
[DRM_CAP_DUMB_PREFER_SHADOW \(C macro\), 775](#)
[DRM_CAP_DUMB_PREFERRED_DEPTH \(C macro\), 775](#)
[DRM_CAP_PAGE_FLIP_TARGET \(C macro\), 777](#)
[DRM_CAP_PRIME \(C macro\), 776](#)
[DRM_CAP_SYNCOBJ \(C macro\), 778](#)
[DRM_CAP_SYNCOBJ_TIMELINE \(C macro\), 778](#)
[DRM_CAP_TIMESTAMP_MONOTONIC \(C macro\), 776](#)
[DRM_CAP_VBLANK_HIGH_CRTC \(C macro\), 775](#)
[drm_class_device_register \(C function\), 774](#)
[drm_class_device_unregister \(C function\), 774](#)
[drm_clflush_pages \(C function\), 174](#)
[drm_clflush_sg \(C function\), 174](#)
[drm_clflush_virt_range \(C function\), 174](#)
[drm_client_buffer \(C struct\), 840](#)
[drm_client_buffer_vmap \(C function\), 843](#)
[drm_client_buffer_vunmap \(C function\), 843](#)
[DRM_CLIENT_CAP_ASPECT_RATIO \(C macro\), 779](#)
[DRM_CLIENT_CAP_ATOMIC \(C macro\), 778](#)
[DRM_CLIENT_CAP_STEREO_3D \(C macro\), 778](#)
[DRM_CLIENT_CAP_UNIVERSAL_PLANES \(C macro\), 778](#)
[DRM_CLIENT_CAP_WRITEBACK_CONNECTORS \(C macro\), 779](#)
[drm_client_dev \(C struct\), 839](#)
[drm_client_dev_hotplug \(C function\), 842](#)
[drm_client_for_each_connector_iter \(C macro\), 841](#)
[drm_client_for_each_modeset \(C macro\), 841](#)

`drm_client_framebuffer_create` (C function), 843

`drm_client_framebuffer_delete` (C function), 844

`drm_client_framebuffer_flush` (C function), 844

`drm_client_funcs` (C struct), 839

`drm_client_init` (C function), 841

`drm_client_modeset_check` (C function), 845

`drm_client_modeset_commit` (C function), 846

`drm_client_modeset_commit_locked` (C function), 845

`drm_client_modeset_dpms` (C function), 846

`drm_client_modeset_probe` (C function), 844

`drm_client_register` (C function), 842

`drm_client_release` (C function), 842

`drm_client_rotation` (C function), 845

`drm_cmdline_mode` (C struct), 360

`drm_color_ctm_s31_32_to_qm_n` (C function), 279

`drm_color_lut_check` (C function), 281

`drm_color_lut_extract` (C function), 281

`drm_color_lut_size` (C function), 282

`drm_color_lut_tests` (C enum), 282

`drm_colorspace` (C enum), 347

`drm_compat_ioctl` (C function), 767

`drm_connector` (C struct), 362

`drm_connector_atomic_hdr_metadata_equal` (C function), 382

`drm_connector_attach_colorspace_property` (C function), 382

`drm_connector_attach_content_protection_property` (C function), 587

`drm_connector_attach_content_type_property` (C function), 376

`drm_connector_attach_dp_subconnector_property` (C function), 376

`drm_connector_attach_edid_property` (C function), 372

`drm_connector_attach_encoder` (C function), 373

`drm_connector_attach_hdr_output_metadata_property` (C function), 382

`drm_connector_attach_max_bpc_property` (C function), 381

`drm_connector_attach_privacy_screen_properties` (C function), 384

`drm_connector_attach_privacy_screen_provider` (C function), 385

`drm_connector_attach_scaling_mode_property` (C function), 379

`drm_connector_attach_tv_margin_properties` (C function), 377

`drm_connector_attach_vrr_capable_property` (C function), 378

`drm_connector_cleanup` (C function), 373

`drm_connector_create_privacy_screen_properties` (C function), 384

`drm_connector_for_each_possible_encoder` (C macro), 370

`drm_connector_funcs` (C struct), 356

`drm_connector_get` (C function), 368

`drm_connector_has_possible_encoder` (C function), 373

`drm_connector_helper_add` (C function), 475

`drm_connector_helper_funcs` (C struct), 470

`drm_connector_helper_get_modes` (C function), 695

`drm_connector_helper_get_modes_fixed` (C function), 694

`drm_connector_helper_get_modes_from_ddc` (C function), 694

`drm_connector_helper_hpd_irq_event` (C function), 693

`drm_connector_helper_tv_get_modes` (C function), 695

`drm_connector_init` (C function), 371

`drm_connector_init_with_ddc` (C function), 371

`drm_connector_is_unregistered` (C function), 369

`drm_connector_list_iter` (C struct), 369

`drm_connector_list_iter_begin` (C function), 374

`drm_connector_list_iter_end` (C function), 375

`drm_connector_list_iter_next` (C function), 375

`drm_connector_list_update` (C function), 339

`drm_connector_lookup` (C function), 368

`drm_connector_oob_hotplug_event` (C function), 385

`drm_connector_put` (C function), 368

`drm_connector_register` (C function), 373

`drm_connector_registration_state` (C enum), 342

`drm_connector_set_link_status_property` (C function), 381

`drm_connector_set_orientation_from_panel` (C function), 384

[drm_connector_set_panel_orientation \(C function\), 383](#)
[drm_connector_set_panel_orientation_with_privacy_screen \(C function\), 383](#)
[drm_connector_set_path_property \(C function\), 380](#)
[drm_connector_set_tile_property \(C function\), 381](#)
[drm_connector_set_vrr_capable_property \(C function\), 383](#)
[drm_connector_state \(C struct\), 354](#)
[drm_connector_status \(C enum\), 342](#)
[drm_connector_tv_margins \(C struct\), 352](#)
[drm_connector_tv_mode \(C enum\), 343](#)
[drm_connector_unregister \(C function\), 374](#)
[drm_connector_update_edid_property \(C function\), 707](#)
[drm_connector_update_privacy_screen \(C function\), 385](#)
[drm_core_check_all_features \(C function\), 22](#)
[drm_core_check_feature \(C function\), 22](#)
[drm_coredump_printer \(C function\), 44](#)
[drm_crtc \(C struct\), 270](#)
[drm_crtc_accurate_vblank_count \(C function\), 446](#)
[drm_crtc_add_crc_entry \(C function\), 769](#)
[drm_crtc_arm_vblank_event \(C function\), 450](#)
[drm_crtc_check_viewport \(C function\), 278](#)
[drm_crtc_cleanup \(C function\), 278](#)
[drm_crtc_commit \(C struct\), 228](#)
[drm_crtc_commit_get \(C function\), 234](#)
[drm_crtc_commit_put \(C function\), 234](#)
[drm_crtc_commit_wait \(C function\), 246](#)
[drm_crtc_create_scaling_filter_property \(C function\), 279](#)
[drm_crtc_enable_color_mgmt \(C function\), 280](#)
[drm_crtc_find \(C function\), 275](#)
[drm_crtc_from_index \(C function\), 276](#)
[drm_crtc_funcs \(C struct\), 263](#)
[drm_crtc_handle_vblank \(C function\), 454](#)
[drm_crtc_helper_add \(C function\), 465](#)
[drm_crtc_helper_atomic_check \(C function\), 743](#)
[drm_crtc_helper_funcs \(C struct\), 459](#)
[drm_crtc_helper_mode_valid_fixed \(C function\), 694](#)
[drm_crtc_helper_set_config \(C function\), 743](#)
[drm_crtc_helper_set_mode \(C function\), 742](#)
[drm_crtc_index \(C function\), 275](#)
[drm_crtc_init \(C function\), 733](#)
[drm_crtc_init_with_planes \(C function\), 276](#)
[drm_crtc_mask \(C function\), 275](#)
[drm_crtc_next_vblank_start \(C function\), 450](#)
[drm_crtc_send_vblank_event \(C function\), 451](#)
[drm_crtc_set_max_vblank_count \(C function\), 453](#)
[drm_crtc_state \(C struct\), 259](#)
[drm_crtc_vblank_count \(C function\), 449](#)
[drm_crtc_vblank_count_and_time \(C function\), 450](#)
[drm_crtc_vblank_get \(C function\), 451](#)
[drm_crtc_vblank_helper_get_vblank_timestamp \(C function\), 448](#)
[drm_crtc_vblank_helper_get_vblank_timestamp_i \(C function\), 448](#)
[drm_crtc_vblank_off \(C function\), 452](#)
[drm_crtc_vblank_on \(C function\), 453](#)
[drm_crtc_vblank_put \(C function\), 452](#)
[drm_crtc_vblank_reset \(C function\), 453](#)
[drm_crtc_vblank_restore \(C function\), 454](#)
[drm_crtc_vblank_waitqueue \(C function\), 447](#)
[drm_crtc_wait_one_vblank \(C function\), 452](#)
[drm_cvt_mode \(C function\), 330](#)
[drm_debug_category \(C enum\), 46](#)
[drm_debug_printer \(C function\), 45](#)
[drm_debugfs_add_file \(C function\), 772](#)
[drm_debugfs_add_files \(C function\), 773](#)
[drm_debugfs_create_files \(C function\), 772](#)
[drm_debugfs_entry \(C struct\), 771](#)
[drm_debugfs_gpuva_info \(C function\), 771](#)
[DRM_DEBUGFS_GPUVA_INFO \(C macro\), 769](#)
[drm_debugfs_info \(C struct\), 770](#)
[drm_default_rgb_quant_range \(C function\), 706](#)
[drm_detect_hdmi_monitor \(C function\), 706](#)
[drm_detect_monitor_audio \(C function\), 706](#)
[drm_dev_alloc \(C function\), 24](#)
[DRM_DEV_DEBUG \(C macro\), 47](#)
[DRM_DEV_DEBUG_DRIVER \(C macro\), 48](#)
[DRM_DEV_DEBUG_KMS \(C macro\), 48](#)
[drm_dev_enter \(C function\), 23](#)
[DRM_DEV_ERROR \(C macro\), 47](#)
[DRM_DEV_ERROR_RATELIMITED \(C macro\), 47](#)
[drm_dev_exit \(C function\), 23](#)

`drm_dev_get` (C function), 24
`drm_dev_has_vblank` (C function), 447
`drm_dev_is_unplugged` (C function), 22
`drm_dev_put` (C function), 25
`drm_dev_register` (C function), 25
`drm_dev_unplug` (C function), 24
`drm_dev_unregister` (C function), 25
`drm_device` (C struct), 12
`drm_display_info` (C struct), 350
`drm_display_info_set_bus_formats` (C function), 375
`drm_display_mode` (C struct), 324
`drm_display_mode_from_cea_vic` (C function), 704
`drm_display_mode_from_videomode` (C function), 332
`drm_display_mode_to_videomode` (C function), 333
`drm_do_get_edid` (C function), 699
`drm_dp_add_payload_part1` (C function), 638
`drm_dp_add_payload_part2` (C function), 639
`drm_dp_atomic_find_time_slots` (C function), 642
`drm_dp_atomic_release_time_slots` (C function), 643
`drm_dp_aux` (C struct), 593
`drm_dp_aux_cec` (C struct), 593
`drm_dp_aux_init` (C function), 607
`drm_dp_aux_msg` (C struct), 592
`drm_dp_aux_register` (C function), 607
`drm_dp_aux_unregister` (C function), 608
`drm_dp_calc_pbn_mode` (C function), 645
`drm_dp_cec_irq` (C function), 618
`drm_dp_cec_register_connector` (C function), 618
`drm_dp_cec_unregister_connector` (C function), 618
`drm_dp_check_act_status` (C function), 645
`drm_dp_desc` (C struct), 596
`drm_dp_downstream_420_passthrough` (C function), 604
`drm_dp_downstream_444_to_420_conversion` (C function), 604
`drm_dp_downstream_debug` (C function), 605
`drm_dp_downstream_id` (C function), 605
`drm_dp_downstream_is_tmds` (C function), 601
`drm_dp_downstream_is_type` (C function), 601
`drm_dp_downstream_max_bpc` (C function), 603
`drm_dp_downstream_max_dotclock` (C function), 602
`drm_dp_downstream_max_tmds_clock` (C function), 603
`drm_dp_downstream_min_tmds_clock` (C function), 603
`drm_dp_downstream_mode` (C function), 605
`drm_dp_downstream_rgb_to_ycbcr_conversion` (C function), 604
`drm_dp_dpcd_probe` (C function), 599
`drm_dp_dpcd_read` (C function), 599
`drm_dp_dpcd_read_link_status` (C function), 600
`drm_dp_dpcd_read_phy_link_status` (C function), 600
`drm_dp_dpcd_readb` (C function), 596
`drm_dp_dpcd_write` (C function), 600
`drm_dp_dpcd_writeb` (C function), 596
`drm_dp_dsc_sink_line_buf_depth` (C function), 609
`drm_dp_dsc_sink_max_slice_count` (C function), 609
`drm_dp_dsc_sink_supported_input_bpcs` (C function), 610
`drm_dp_dsc_sink_supports_format` (C function), 592
`drm_dp_dual_mode_detect` (C function), 620
`drm_dp_dual_mode_get_tmds_output` (C function), 621
`drm_dp_dual_mode_max_tmds_clock` (C function), 621
`drm_dp_dual_mode_read` (C function), 619
`drm_dp_dual_mode_set_tmds_output` (C function), 621
`drm_dp_dual_mode_type` (C enum), 619
`drm_dp_dual_mode_write` (C function), 620
`drm_dp_get_dual_mode_type_name` (C function), 622
`drm_dp_get_pcon_max_frl_bw` (C function), 612
`drm_dp_get_phy_test_pattern` (C function), 612
`drm_dp_get_vc_payload_bw` (C function), 639
`drm_dp_has_quirk` (C function), 597
`drm_dp_lttpr_count` (C function), 611
`drm_dp_lttpr_max_lane_count` (C function), 611
`drm_dp_lttpr_max_link_rate` (C function), 611
`drm_dp_lttpr_pre_emphasis_level_3_supported` (C function), 612

[drm_dp_lttpr_voltage_swing_level_3_support \(C function\), 611](#)
[drm_dp_mst_add_affected_dsc_crcs \(C function\), 646](#)
[drm_dp_mst_atomic_check \(C function\), 647](#)
[drm_dp_mst_atomic_enable_dsc \(C function\), 646](#)
[drm_dp_mst_atomic_payload \(C struct\), 630](#)
[drm_dp_mst_atomic_setup_commit \(C function\), 644](#)
[drm_dp_mst_atomic_wait_for_dependencies \(C function\), 644](#)
[drm_dp_mst_branch \(C struct\), 628](#)
[drm_dp_mst_connector_early_unregister \(C function\), 637](#)
[drm_dp_mst_connector_late_register \(C function\), 637](#)
[drm_dp_mst_detect_port \(C function\), 641](#)
[drm_dp_mst_dsc_aux_for_port \(C function\), 649](#)
[drm_dp_mst_dump_topology \(C function\), 646](#)
[drm_dp_mst_edid_read \(C function\), 642](#)
[drm_dp_mst_get_edid \(C function\), 642](#)
[drm_dp_mst_get_mstb_malloc \(C function\), 649](#)
[drm_dp_mst_get_port_malloc \(C function\), 636](#)
[drm_dp_mst_hpd_irq_handle_event \(C function\), 641](#)
[drm_dp_mst_hpd_irq_send_new_request \(C function\), 641](#)
[drm_dp_mst_port \(C struct\), 626](#)
[drm_dp_mst_put_mstb_malloc \(C function\), 650](#)
[drm_dp_mst_put_port_malloc \(C function\), 637](#)
[drm_dp_mst_root_conn_atomic_check \(C function\), 644](#)
[drm_dp_mst_topology_get_mstb \(C function\), 650](#)
[drm_dp_mst_topology_get_port \(C function\), 651](#)
[drm_dp_mst_topology_mgr \(C struct\), 632](#)
[drm_dp_mst_topology_mgr_destroy \(C function\), 649](#)
[drm_dp_mst_topology_mgr_init \(C function\), 648](#)
[drm_dp_mst_topology_mgr_resume \(C function\), 640](#)
[drm_dp_mst_topology_mgr_set_mst \(C function\), 640](#)
[drm_dp_mst_topology_mgr_suspend \(C function\), 640](#)
[drm_dp_mst_topology_put_mstb \(C function\), 650](#)
[drm_dp_mst_topology_put_port \(C function\), 651](#)
[drm_dp_mst_topology_state \(C struct\), 631](#)
[drm_dp_mst_topology_try_get_mstb \(C function\), 650](#)
[drm_dp_mst_topology_try_get_port \(C function\), 651](#)
[drm_dp_mst_update_slots \(C function\), 645](#)
[drm_dp_pcon_frl_configure_1 \(C function\), 613](#)
[drm_dp_pcon_frl_configure_2 \(C function\), 613](#)
[drm_dp_pcon_frl_enable \(C function\), 614](#)
[drm_dp_pcon_frl_prepare \(C function\), 613](#)
[drm_dp_pcon_hdmi_frl_link_error_count \(C function\), 615](#)
[drm_dp_pcon_hdmi_link_active \(C function\), 614](#)
[drm_dp_pcon_hdmi_link_mode \(C function\), 614](#)
[drm_dp_pcon_is_frl_ready \(C function\), 613](#)
[drm_dp_pcon_pps_default \(C function\), 615](#)
[drm_dp_pcon_pps_override_buf \(C function\), 615](#)
[drm_dp_pcon_reset_frl_config \(C function\), 614](#)
[drm_dp_phy_name \(C function\), 599](#)
[drm_dp_phy_test_params \(C struct\), 598](#)
[drm_dp_psr_setup_time \(C function\), 608](#)
[drm_dp_quirk \(C enum\), 597](#)
[drm_dp_read_desc \(C function\), 609](#)
[drm_dp_read_downstream_info \(C function\), 602](#)
[drm_dp_read_dpcd_caps \(C function\), 602](#)
[drm_dp_read_lttpr_common_caps \(C function\), 610](#)
[drm_dp_read_lttpr_phy_caps \(C function\), 610](#)
[drm_dp_read_mst_cap \(C function\), 639](#)
[drm_dp_read_sink_count \(C function\), 607](#)
[drm_dp_read_sink_count_cap \(C function\), 606](#)
[drm_dp_remote_aux_init \(C function\), 607](#)
[drm_dp_remove_payload \(C function\), 638](#)
[drm_dp_send_real_edid_checksum \(C function\), 601](#)
[drm_dp_set_phy_test_pattern \(C function\),](#)

- 612
- `drm_dp_set_subconnector_property` (C function), 606
- `drm_dp_start_crc` (C function), 608
- `drm_dp_stop_crc` (C function), 608
- `drm_dp_subconnector_type` (C function), 606
- `drm_dp_vsc_sdp` (C struct), 591
- `drm_driver` (C struct), 17
- `drm_driver_feature` (C enum), 16
- `drm_driver_legacy_fb_format` (C function), 294
- `drm_drv_uses_atomic_modeset` (C function), 23
- `drm_dsc_compute_rc_parameters` (C function), 688
- `drm_dsc_config` (C struct), 678
- `drm_dsc_dp_pps_header_init` (C function), 686
- `drm_dsc_dp_rc_buffer_size` (C function), 686
- `drm_dsc_flatness_det_thresh` (C function), 688
- `drm_dsc_get_bpp_int` (C function), 688
- `drm_dsc_initial_scale_value` (C function), 688
- `drm_dsc_picture_parameter_set` (C struct), 682
- `drm_dsc_pps_infoframe` (C struct), 686
- `drm_dsc_pps_payload_pack` (C function), 687
- `drm_dsc_rc_range_parameters` (C struct), 678
- `drm_dsc_set_const_params` (C function), 687
- `drm_dsc_set_rc_buf_thresh` (C function), 687
- `drm_dsc_setup_rc_params` (C function), 687
- `drm_edid_alloc` (C function), 700
- `drm_edid_are_equal` (C function), 698
- `drm_edid_block_valid` (C function), 698
- `drm_edid_connector_add_modes` (C function), 707
- `drm_edid_connector_update` (C function), 707
- `drm_edid_decode_mfg_id` (C function), 697
- `drm_edid_decode_panel_id` (C function), 697
- `drm_edid_dup` (C function), 700
- `drm_edid_duplicate` (C function), 704
- `drm_edid_encode_panel_id` (C macro), 697
- `drm_edid_free` (C function), 701
- `drm_edid_get_monitor_name` (C function), 705
- `drm_edid_get_panel_id` (C function), 703
- `drm_edid_header_is_valid` (C function), 698
- `drm_edid_is_valid` (C function), 698
- `drm_edid_override_connector_update` (C function), 699
- `drm_edid_raw` (C function), 700
- `drm_edid_read` (C function), 702
- `drm_edid_read_custom` (C function), 701
- `drm_edid_read_ddc` (C function), 702
- `drm_edid_read_switcheroo` (C function), 704
- `drm_edid_to_sad` (C function), 705
- `drm_edid_to_speaker_allocation` (C function), 705
- `drm_edid_valid` (C function), 699
- `drm_edp_backlight_disable` (C function), 616
- `drm_edp_backlight_enable` (C function), 616
- `drm_edp_backlight_info` (C struct), 597
- `drm_edp_backlight_init` (C function), 617
- `drm_edp_backlight_set_level` (C function), 615
- `drm_edp_backlight_supported` (C function), 592
- `drm_eld_calc_baseline_block_size` (C function), 696
- `drm_eld_get_conn_type` (C function), 696
- `drm_eld_get_spk_alloc` (C function), 696
- `drm_eld_mnl` (C function), 695
- `drm_eld_sad` (C function), 695
- `drm_eld_sad_count` (C function), 696
- `drm_eld_size` (C function), 696
- `drm_encoder` (C struct), 393
- `drm_encoder_cleanup` (C function), 398
- `drm_encoder_crtc_ok` (C function), 396
- `drm_encoder_find` (C function), 396
- `drm_encoder_funcs` (C struct), 392
- `drm_encoder_helper_add` (C function), 470
- `drm_encoder_helper_funcs` (C struct), 466
- `drm_encoder_index` (C function), 396
- `drm_encoder_init` (C function), 397
- `drm_encoder_mask` (C function), 396
- `drm_err_printer` (C function), 46
- `drm_event` (C struct), 781
- `drm_event_cancel_free` (C function), 40
- `DRM_EVENT_CRTC_SEQUENCE` (C macro), 782
- `DRM_EVENT_FLIP_COMPLETE` (C macro), 782
- `drm_event_reserve_init` (C function), 40
- `drm_event_reserve_init_locked` (C function), 39
- `DRM_EVENT_VBLANK` (C macro), 782
- `drm_exec` (C struct), 182
- `drm_exec_cleanup` (C function), 185

[drm_exec_fini \(C function\), 184](#)
[drm_exec_for_each_locked_object \(C macro\), 183](#)
[drm_exec_for_each_locked_object_reverse \(C macro\), 183](#)
[drm_exec_init \(C function\), 184](#)
[drm_exec_is_contended \(C function\), 184](#)
[drm_exec_lock_obj \(C function\), 185](#)
[drm_exec_obj \(C function\), 182](#)
[drm_exec_prepare_array \(C function\), 186](#)
[drm_exec_prepare_obj \(C function\), 186](#)
[drm_exec_retry_on_contention \(C macro\), 184](#)
[drm_exec_unlock_obj \(C function\), 185](#)
[drm_exec_until_all_locked \(C macro\), 183](#)
[drm_fb_blit \(C function\), 542](#)
[drm_fb_build_fourcc_list \(C function\), 543](#)
[drm_fb_clip_offset \(C function\), 535](#)
[drm_fb_dma_get_gem_addr \(C function\), 545](#)
[drm_fb_dma_get_gem_obj \(C function\), 544](#)
[drm_fb_dma_sync_non_coherent \(C function\), 545](#)
[drm_fb_helper \(C struct\), 525](#)
[drm_fb_helper_alloc_info \(C function\), 529](#)
[drm_fb_helper_blank \(C function\), 528](#)
[drm_fb_helper_check_var \(C function\), 531](#)
[drm_fb_helper_debug_enter \(C function\), 527](#)
[drm_fb_helper_debug_leave \(C function\), 527](#)
[DRM_FB_HELPER_DEFAULT_OPS \(C macro\), 527](#)
[drm_fb_helper_deferred_io \(C function\), 530](#)
[drm_fb_helper_fill_info \(C function\), 532](#)
[drm_fb_helper_fini \(C function\), 530](#)
[drm_fb_helper_funcs \(C struct\), 525](#)
[drm_fb_helper_hotplug_event \(C function\), 533](#)
[drm_fb_helper_init \(C function\), 529](#)
[drm_fb_helper_initial_config \(C function\), 532](#)
[drm_fb_helper_ioctl \(C function\), 531](#)
[drm_fb_helper_lastclose \(C function\), 533](#)
[drm_fb_helper_output_poll_changed \(C function\), 534](#)
[drm_fb_helper_pan_display \(C function\), 532](#)
[drm_fb_helper_prepare \(C function\), 528](#)
[drm_fb_helper_release_info \(C function\), 529](#)
[drm_fb_helper_restore_fbdev_mode_unlocked \(C function\), 528](#)
[drm_fb_helper_set_par \(C function\), 532](#)
[drm_fb_helper_set_suspend \(C function\), 530](#)
[drm_fb_helper_set_suspend_unlocked \(C function\), 530](#)
[drm_fb_helper_setcmap \(C function\), 531](#)
[drm_fb_helper_surface_size \(C struct\), 524](#)
[drm_fb_helper_unprepare \(C function\), 528](#)
[drm_fb_helper_unregister_info \(C function\), 530](#)
[drm_fb_memcpy \(C function\), 535](#)
[drm_fb_swab \(C function\), 535](#)
[drm_fb_xrgb8888_to_argb1555 \(C function\), 538](#)
[drm_fb_xrgb8888_to_argb2101010 \(C function\), 541](#)
[drm_fb_xrgb8888_to_argb8888 \(C function\), 540](#)
[drm_fb_xrgb8888_to_gray8 \(C function\), 541](#)
[drm_fb_xrgb8888_to_mono \(C function\), 543](#)
[drm_fb_xrgb8888_to_rgb332 \(C function\), 536](#)
[drm_fb_xrgb8888_to_rgb565 \(C function\), 537](#)
[drm_fb_xrgb8888_to_rgb888 \(C function\), 539](#)
[drm_fb_xrgb8888_to_rgba5551 \(C function\), 538](#)
[drm_fb_xrgb8888_to_xrgb1555 \(C function\), 537](#)
[drm_fb_xrgb8888_to_xrgb2101010 \(C function\), 540](#)
[drm_fbdev_generic_setup \(C function\), 534](#)
[drm_file \(C struct\), 33](#)
[drm_file_get_master \(C function\), 755](#)
[drm_flip_task \(C struct\), 730](#)
[drm_flip_work \(C struct\), 730](#)
[drm_flip_work_allocate_task \(C function\), 731](#)
[drm_flip_work_cleanup \(C function\), 732](#)
[drm_flip_work_commit \(C function\), 731](#)
[drm_flip_work_init \(C function\), 732](#)
[drm_flip_work_queue \(C function\), 731](#)
[drm_flip_work_queue_task \(C function\), 731](#)
[drm_for_each_bridge_in_chain \(C macro\), 565](#)
[drm_for_each_connector_iter \(C macro\), 370](#)
[drm_for_each_crtc \(C macro\), 275](#)
[drm_for_each_crtc_reverse \(C macro\), 276](#)

`drm_for_each_encoder` (*C macro*), 397
`drm_for_each_encoder_mask` (*C macro*), 397
`drm_for_each_legacy_plane` (*C macro*), 310
`drm_for_each_plane` (*C macro*), 310
`drm_for_each_plane_mask` (*C macro*), 309
`drm_for_each_privobj` (*C macro*), 232
`drm_format_info` (*C function*), 295
`drm_format_info` (*C struct*), 291
`drm_format_info_block_height` (*C function*), 296
`drm_format_info_block_width` (*C function*), 295
`drm_format_info_bpp` (*C function*), 296
`drm_format_info_is_yuv_packed` (*C function*), 292
`drm_format_info_is_yuv_planar` (*C function*), 292
`drm_format_info_is_yuv_sampling_410` (*C function*), 293
`drm_format_info_is_yuv_sampling_411` (*C function*), 293
`drm_format_info_is_yuv_sampling_420` (*C function*), 293
`drm_format_info_is_yuv_sampling_422` (*C function*), 293
`drm_format_info_is_yuv_sampling_444` (*C function*), 293
`drm_format_info_is_yuv_semiplanar` (*C function*), 292
`drm_format_info_min_pitch` (*C function*), 296
`drm_format_info_plane_height` (*C function*), 294
`drm_format_info_plane_width` (*C function*), 294
`DRM_FORMAT_MAX_PLANES` (*C macro*), 291
`drm_framebuffer` (*C struct*), 284
`drm_framebuffer_assign` (*C function*), 286
`drm_framebuffer_cleanup` (*C function*), 288
`drm_framebuffer_funcs` (*C struct*), 283
`drm_framebuffer_get` (*C function*), 285
`drm_framebuffer_init` (*C function*), 287
`drm_framebuffer_lookup` (*C function*), 287
`drm_framebuffer_plane_height` (*C function*), 289
`drm_framebuffer_plane_width` (*C function*), 289
`drm_framebuffer_put` (*C function*), 285
`drm_framebuffer_read_refcount` (*C function*), 286
`drm_framebuffer_remove` (*C function*), 288
`drm_framebuffer_unregister_private` (*C function*), 288
`drm_gem_begin_shadow_fb_access` (*C function*), 515
`drm_gem_create_mmap_offset` (*C function*), 85
`drm_gem_create_mmap_offset_size` (*C function*), 85
`drm_gem_destroy_shadow_plane_state` (*C function*), 515
`drm_gem_dma_create` (*C function*), 94
`DRM_GEM_DMA_DRIVER_OPS` (*C macro*), 93
`DRM_GEM_DMA_DRIVER_OPS_VMAP` (*C macro*), 94
`DRM_GEM_DMA_DRIVER_OPS_VMAP_WITH_DUMB_CREATE` (*C macro*), 93
`DRM_GEM_DMA_DRIVER_OPS_WITH_DUMB_CREATE` (*C macro*), 93
`drm_gem_dma_dumb_create` (*C function*), 95
`drm_gem_dma_dumb_create_internal` (*C function*), 95
`drm_gem_dma_free` (*C function*), 95
`drm_gem_dma_get_sg_table` (*C function*), 97
`drm_gem_dma_get_unmapped_area` (*C function*), 96
`drm_gem_dma_mmap` (*C function*), 98
`drm_gem_dma_object` (*C struct*), 91
`drm_gem_dma_object_free` (*C function*), 92
`drm_gem_dma_object_get_sg_table` (*C function*), 92
`drm_gem_dma_object_mmap` (*C function*), 92
`drm_gem_dma_object_print_info` (*C function*), 92
`drm_gem_dma_prime_import_sg_table` (*C function*), 97
`drm_gem_dma_prime_import_sg_table_vmap` (*C function*), 98
`drm_gem_dma_print_info` (*C function*), 96
`drm_gem_dma_resv_wait` (*C function*), 87
`drm_gem_dma_vmap` (*C function*), 97
`drm_gem_dmabuf_export` (*C function*), 124
`drm_gem_dmabuf_mmap` (*C function*), 128
`drm_gem_dmabuf_release` (*C function*), 125
`drm_gem_dmabuf_vmap` (*C function*), 127
`drm_gem_dmabuf_vunmap` (*C function*), 127
`drm_gem_dumb_map_offset` (*C function*), 83
`drm_gem_duplicate_shadow_plane_state` (*C function*), 514
`drm_gem_end_shadow_fb_access` (*C function*), 516
`drm_gem_evict` (*C function*), 91
`drm_gem_fb_afbc_init` (*C function*), 550

`drm_gem_fb_begin_cpu_access` (C function), 549
`drm_gem_fb_create` (C function), 547
`drm_gem_fb_create_handle` (C function), 546
`drm_gem_fb_create_with_dirty` (C function), 548
`drm_gem_fb_create_with_funcs` (C function), 547
`drm_gem_fb_destroy` (C function), 546
`drm_gem_fb_end_cpu_access` (C function), 550
`drm_gem_fb_get_obj` (C function), 545
`drm_gem_fb_init_with_funcs` (C function), 546
`drm_gem_fb_vmap` (C function), 549
`drm_gem_fb_vunmap` (C function), 549
`DRM_GEM_FOPS` (C macro), 80
`drm_gem_for_each_gpuva` (C macro), 82
`drm_gem_for_each_gpuva_safe` (C macro), 82
`drm_gem_free_mmap_offset` (C function), 84
`drm_gem_get_pages` (C function), 85
`drm_gem_gpuva_init` (C function), 81
`drm_gem_gpuva_set_lock` (C macro), 81
`drm_gem_handle_create` (C function), 84
`drm_gem_handle_delete` (C function), 83
`drm_gem_lock_reservations` (C function), 89
`drm_gem_lru` (C struct), 77
`drm_gem_lru_init` (C function), 89
`drm_gem_lru_move_tail` (C function), 90
`drm_gem_lru_move_tail_locked` (C function), 90
`drm_gem_lru_remove` (C function), 90
`drm_gem_lru_scan` (C function), 90
`drm_gem_map_attach` (C function), 126
`drm_gem_map_detach` (C function), 126
`drm_gem_map_dma_buf` (C function), 126
`drm_gem_mmap` (C function), 89
`drm_gem_mmap_obj` (C function), 88
`drm_gem_object` (C struct), 78
`drm_gem_object_free` (C function), 87
`drm_gem_object_funcs` (C struct), 76
`drm_gem_object_get` (C function), 81
`drm_gem_object_init` (C function), 82
`drm_gem_object_is_shared_for_memory_stats` (C function), 81
`drm_gem_object_lookup` (C function), 86
`drm_gem_object_put` (C function), 81
`drm_gem_object_release` (C function), 87
`drm_gem_object_status` (C enum), 75
`drm_gem_objects_lookup` (C function), 86
`drm_gem_plane_helper_prepare_fb` (C function), 513
`drm_gem_prime_export` (C function), 129
`drm_gem_prime_fd_to_handle` (C function), 125
`drm_gem_prime_handle_to_fd` (C function), 125
`drm_gem_prime_import` (C function), 129
`drm_gem_prime_import_dev` (C function), 129
`drm_gem_prime_mmap` (C function), 127
`drm_gem_private_object_fini` (C function), 83
`drm_gem_private_object_init` (C function), 83
`drm_gem_put_pages` (C function), 86
`drm_gem_reset_shadow_plane` (C function), 515
`DRM_GEM_SHADOW_PLANE_FUNCS` (C macro), 513
`DRM_GEM_SHADOW_PLANE_HELPER_FUNCS` (C macro), 513
`drm_gem_shmem_create` (C function), 102
`DRM_GEM_SHMEM_DRIVER_OPS` (C macro), 101
`drm_gem_shmem_dumb_create` (C function), 103
`drm_gem_shmem_free` (C function), 102
`drm_gem_shmem_get_pages_sgt` (C function), 104
`drm_gem_shmem_get_sg_table` (C function), 104
`drm_gem_shmem_mmap` (C function), 103
`drm_gem_shmem_object` (C struct), 99
`drm_gem_shmem_object_free` (C function), 100
`drm_gem_shmem_object_get_sg_table` (C function), 101
`drm_gem_shmem_object_mmap` (C function), 101
`drm_gem_shmem_object_pin` (C function), 100
`drm_gem_shmem_object_print_info` (C function), 100
`drm_gem_shmem_object_unpin` (C function), 101
`drm_gem_shmem_pin` (C function), 102
`drm_gem_shmem_prime_import_sg_table` (C function), 104
`drm_gem_shmem_print_info` (C function), 103
`drm_gem_shmem_unpin` (C function), 102
`DRM_GEM_SIMPLE_DISPLAY_PIPE_SHADOW_PLANE_FUNCS` (C macro), 513
`drm_gem_simple_kms_begin_shadow_fb_access` (C function), 516
`drm_gem_simple_kms_destroy_shadow_plane_state`

(C function), 517
drm_gem_simple_kms_duplicate_shadow_plane_state (C function), 517
drm_gem_simple_kms_end_shadow_fb_access (C function), 516
drm_gem_simple_kms_reset_shadow_plane (C function), 517
drm_gem_ttm_dumb_map_offset (C function), 115
drm_gem_ttm_mmap (C function), 115
drm_gem_ttm_print_info (C function), 114
drm_gem_ttm_vmap (C function), 114
drm_gem_ttm_vunmap (C function), 114
drm_gem_unmap_dma_buf (C function), 127
drm_gem_vm_close (C function), 88
drm_gem_vm_open (C function), 88
drm_gem_vram_create (C function), 108
DRM_GEM_VRAM_DRIVER (C macro), 107
drm_gem_vram_driver_dumb_create (C function), 111
drm_gem_vram_fill_create_dumb (C function), 110
drm_gem_vram_object (C struct), 106
drm_gem_vram_of_bo (C function), 107
drm_gem_vram_of_gem (C function), 107
drm_gem_vram_offset (C function), 109
drm_gem_vram_pin (C function), 109
drm_gem_vram_plane_helper_cleanup_fb (C function), 112
DRM_GEM_VRAM_PLANE_HELPER_FUNCS (C macro), 107
drm_gem_vram_plane_helper_prepare_fb (C function), 111
drm_gem_vram_put (C function), 109
drm_gem_vram_simple_display_pipe_cleanup (C function), 112
drm_gem_vram_simple_display_pipe_prepare (C function), 112
drm_gem_vram_unpin (C function), 110
drm_gem_vram_vmap (C function), 110
drm_gem_vram_vunmap (C function), 110
drm_get_buddy (C function), 171
drm_get_connector_status_name (C function), 374
drm_get_connector_type_name (C function), 370
drm_get_edid (C function), 701
drm_get_edid_switcheroo (C function), 703
drm_get_format_info (C function), 295
drm_get_panel_orientation_quirk (C function), 584
drm_get_subpixel_order_name (C function), 375
drm_get_tv_mode_from_name (C function), 376
drm_gpu_scheduler (C struct), 193
drm_gpuva (C struct), 153
drm_gpuva_find (C function), 165
drm_gpuva_find_first (C function), 164
drm_gpuva_find_next (C function), 165
drm_gpuva_find_prev (C function), 165
drm_gpuva_first_op (C macro), 161
drm_gpuva_flags (C enum), 153
drm_gpuva_fn_ops (C struct), 161
drm_gpuva_for_each_op (C macro), 160
drm_gpuva_for_each_op_from_reverse (C macro), 160
drm_gpuva_for_each_op_safe (C macro), 160
drm_gpuva_for_each_va (C macro), 156
drm_gpuva_for_each_va_range (C macro), 155
drm_gpuva_for_each_va_range_safe (C macro), 155
drm_gpuva_for_each_va_safe (C macro), 156
drm_gpuva_gem_unmap_ops_create (C function), 170
drm_gpuva_insert (C function), 163
drm_gpuva_interval_empty (C function), 166
drm_gpuva_invalidate (C function), 154
drm_gpuva_invalidated (C function), 154
drm_gpuva_last_op (C macro), 161
drm_gpuva_link (C function), 164
drm_gpuva_manager (C struct), 154
drm_gpuva_manager_destroy (C function), 163
drm_gpuva_manager_init (C function), 162
drm_gpuva_map (C function), 166
drm_gpuva_next_op (C macro), 161
drm_gpuva_op (C struct), 159
drm_gpuva_op_map (C struct), 157
drm_gpuva_op_prefetch (C struct), 158
drm_gpuva_op_remap (C struct), 158
drm_gpuva_op_type (C enum), 156
drm_gpuva_op_unmap (C struct), 157
drm_gpuva_ops (C struct), 160
drm_gpuva_ops_free (C function), 171
drm_gpuva_prefetch_ops_create (C function), 170
drm_gpuva_prev_op (C macro), 161
drm_gpuva_remap (C function), 166
drm_gpuva_remove (C function), 164
drm_gpuva_sm_map (C function), 167

drm_gpuva_sm_map_ops_create (C function), 168
 drm_gpuva_sm_unmap (C function), 167
 drm_gpuva_sm_unmap_ops_create (C function), 169
 drm_gpuva_unlink (C function), 164
 drm_gpuva_unmap (C function), 166
 drm_gtf_mode (C function), 332
 drm_gtf_mode_complex (C function), 331
 drm_handle_vblank (C function), 454
 drm_hdcp_check_ksvs_revoked (C function), 586
 drm_hdcp_update_content_protection (C function), 587
 drm_hdmi_avi_infoframe_from_display_mode (C function), 709
 drm_hdmi_avi_infoframe_quant_range (C function), 709
 drm_hdmi_dsc_cap (C struct), 344
 drm_hdmi_info (C struct), 345
 drm_hdmi_vendor_infoframe_from_display_mode (C function), 709
 drm_helper_connector_dpms (C function), 744
 drm_helper_crtc_in_use (C function), 741
 drm_helper_disable_unused_functions (C function), 742
 drm_helper_encoder_in_use (C function), 741
 drm_helper_force_disable_all (C function), 745
 drm_helper_hpd_irq_event (C function), 693
 drm_helper_mode_fill_fb_struct (C function), 733
 drm_helper_move_panel_connectors_to_head (C function), 732
 drm_helper_probe_detect (C function), 689
 drm_helper_probe_single_connector_modes (C function), 690
 drm_helper_resume_force_mode (C function), 744
 drm_i915_engine_info (C struct), 823
 drm_i915_gem_caching (C struct), 809
 drm_i915_gem_context_create_ext (C struct), 810
 drm_i915_gem_context_create_ext_setparam (C struct), 816
 drm_i915_gem_context_param (C struct), 811
 drm_i915_gem_create_ext (C struct), 828
 drm_i915_gem_create_ext_memory_regions (C struct), 830
 drm_i915_gem_create_ext_protected_content (C struct), 832
 drm_i915_gem_create_ext_set_pat (C struct), 832
 drm_i915_gem_create_ext_vm_private (C struct), 1212
 drm_i915_gem_engine_class (C enum), 802
 drm_i915_gem_exec_fence (C struct), 806
 drm_i915_gem_execbuffer2 (C struct), 807
 drm_i915_gem_execbuffer3 (C struct), 1211
 drm_i915_gem_execbuffer_ext_timeline_fences (C struct), 807
 drm_i915_gem_memory_class (C enum), 825
 drm_i915_gem_memory_class_instance (C struct), 825
 drm_i915_gem_mmap_offset (C struct), 804
 drm_i915_gem_set_domain (C struct), 805
 drm_i915_gem_timeline_fence (C struct), 1208
 drm_i915_gem_userptr (C struct), 817
 drm_i915_gem_vm_bind (C struct), 1209
 drm_i915_gem_vm_control (C struct), 816
 drm_i915_gem_vm_unbind (C struct), 1210
 drm_i915_getparam (C struct), 803
 drm_i915_getparam_t (C type), 804
 drm_i915_memory_region_info (C struct), 825
 drm_i915_perf_oa_config (C struct), 818
 drm_i915_query (C struct), 820
 drm_i915_query_engine_info (C struct), 824
 drm_i915_query_item (C struct), 818
 drm_i915_query_memory_regions (C struct), 827
 drm_i915_query_perf_config (C struct), 824
 drm_i915_query_topology_info (C struct), 820
 drm_info_list (C struct), 769
 drm_info_node (C struct), 770
 drm_info_printer (C function), 45
 drm_invalid_op (C function), 765
 drm_ioctl (C function), 766
 drm_ioctl_compat_t (C macro), 763
 DRM_IOCTL_DEF_DRV (C macro), 765
 drm_ioctl_desc (C struct), 764
 drm_ioctl_flags (C enum), 764
 drm_ioctl_flags (C function), 766
 DRM_IOCTL_GEM_CLOSE (C macro), 780
 DRM_IOCTL_MODE_GETFB2 (C macro), 781
 DRM_IOCTL_MODE_RMFB (C macro), 781
 DRM_IOCTL_PRIME_FD_TO_HANDLE (C macro), 780

`DRM_IOCTL_PRIME_HANDLE_TO_FD` (C macro), 780

`drm_ioctl_t` (C macro), 763

`drm_is_accel_client` (C function), 36

`drm_is_current_master` (C function), 754

`drm_is_panel_follower` (C function), 582

`drm_is_primary_client` (C function), 36

`drm_is_render_client` (C function), 36

`drm_kms_helper_connector_hotplug_event` (C function), 691

`drm_kms_helper_hotplug_event` (C function), 691

`drm_kms_helper_is_poll_worker` (C function), 692

`drm_kms_helper_poll_disable` (C function), 692

`drm_kms_helper_poll_enable` (C function), 689

`drm_kms_helper_poll_fini` (C function), 693

`drm_kms_helper_poll_init` (C function), 692

`drm_kms_helper_poll_reschedule` (C function), 689

`drm_legacy_pci_exit` (C function), 30

`drm_legacy_pci_init` (C function), 30

`drm_link_status` (C enum), 345

`drm_lspcon_get_mode` (C function), 622

`drm_lspcon_mode` (C enum), 619

`drm_lspcon_set_mode` (C function), 622

`drm_luminance_range_info` (C struct), 346

`drm_master` (C struct), 755

`drm_master_get` (C function), 754

`drm_master_put` (C function), 755

`drm_match_cea_mode` (C function), 704

`drm_memcpy_from_wc` (C function), 174

`drm_memory_stats` (C struct), 37

`drm_minor` (C struct), 31

`DRM_MIPI_DBI_SIMPLE_DISPLAY_PIPE_FUNCS` (C macro), 654

`drm_mm` (C struct), 133

`drm_mm_clean` (C function), 138

`drm_mm_for_each_hole` (C macro), 137

`drm_mm_for_each_node` (C macro), 136

`drm_mm_for_each_node_in_range` (C macro), 138

`drm_mm_for_each_node_safe` (C macro), 136

`drm_mm_hole_follows` (C function), 135

`drm_mm_hole_node_end` (C function), 135

`drm_mm_hole_node_start` (C function), 135

`drm_mm_init` (C function), 142

`drm_mm_initialized` (C function), 134

`drm_mm_insert_mode` (C enum), 132

`drm_mm_insert_node` (C function), 138

`drm_mm_insert_node_generic` (C function), 137

`drm_mm_insert_node_in_range` (C function), 140

`drm_mm_node` (C struct), 133

`drm_mm_node_allocated` (C function), 134

`drm_mm_nodes` (C macro), 136

`drm_mm_print` (C function), 143

`drm_mm_remove_node` (C function), 140

`drm_mm_replace_node` (C function), 140

`drm_mm_reserve_node` (C function), 139

`drm_mm_scan` (C struct), 134

`drm_mm_scan_add_block` (C function), 141

`drm_mm_scan_color_evict` (C function), 142

`drm_mm_scan_init` (C function), 139

`drm_mm_scan_init_with_range` (C function), 141

`drm_mm_scan_remove_block` (C function), 142

`drm_mm_takedown` (C function), 143

`DRM_MODE_ARG` (C macro), 328

`DRM_MODE_ATOMIC_ALLOW_MODESET` (C macro), 792

`DRM_MODE_ATOMIC_FLAGS` (C macro), 792

`DRM_MODE_ATOMIC_NONBLOCK` (C macro), 792

`DRM_MODE_ATOMIC_TEST_ONLY` (C macro), 792

`drm_mode_config` (C struct), 212

`drm_mode_config_cleanup` (C function), 221

`drm_mode_config_funcs` (C struct), 208

`drm_mode_config_helper_funcs` (C struct), 479

`drm_mode_config_helper_resume` (C function), 734

`drm_mode_config_helper_suspend` (C function), 734

`drm_mode_config_init` (C function), 221

`drm_mode_config_reset` (C function), 221

`drm_mode_copy` (C function), 335

`drm_mode_create` (C function), 329

`drm_mode_create_aspect_ratio_property` (C function), 379

`drm_mode_create_blob` (C struct), 793

`drm_mode_create_content_type_property` (C function), 380

`drm_mode_create_dp_colorspace_property` (C function), 379

`drm_mode_create_dvi_i_properties` (C function), 376

`drm_mode_create_from_cmdline_mode` (C function), 340

`drm_mode_create_hdmi_colorspace_property`

- (C function), 379
- drm_mode_create_lease (C struct), 793
- drm_mode_create_scaling_mode_property (C function), 378
- drm_mode_create_suggested_offset_properties (C function), 380
- drm_mode_create_tile_group (C function), 386
- drm_mode_create_tv_margin_properties (C function), 377
- drm_mode_create_tv_properties (C function), 378
- drm_mode_create_tv_properties_legacy (C function), 377
- drm_mode_crtc_set_gamma_size (C function), 280
- drm_mode_debug_printmodeline (C function), 328
- drm_mode_destroy (C function), 329
- drm_mode_destroy_blob (C struct), 793
- drm_mode_duplicate (C function), 336
- drm_mode_equal (C function), 336
- drm_mode_equal_no_clocks (C function), 337
- drm_mode_equal_no_clocks_no_stereo (C function), 337
- drm_mode_fb_cmd2 (C struct), 788
- DRM_MODE_FMT (C macro), 328
- drm_mode_get_connector (C struct), 785
- drm_mode_get_hv_timing (C function), 335
- drm_mode_get_lease (C struct), 795
- drm_mode_get_plane (C struct), 784
- drm_mode_get_property (C struct), 787
- drm_mode_get_tile_group (C function), 386
- drm_mode_init (C function), 335
- DRM_MODE_INIT (C macro), 323
- drm_mode_is_420 (C function), 341
- drm_mode_is_420_also (C function), 341
- drm_mode_is_420_only (C function), 340
- drm_mode_is_stereo (C function), 328
- drm_mode_legacy_fb_format (C function), 294
- drm_mode_list_lessees (C struct), 794
- drm_mode_match (C function), 336
- drm_mode_modeinfo (C struct), 783
- drm_mode_object (C struct), 222
- drm_mode_object_find (C function), 224
- drm_mode_object_get (C function), 224
- drm_mode_object_put (C function), 224
- DRM_MODE_PAGE_FLIP_ASYNC (C macro), 791
- DRM_MODE_PAGE_FLIP_EVENT (C macro), 791
- DRM_MODE_PAGE_FLIP_FLAGS (C macro), 791
- drm_mode_parse_command_line_for_connector (C function), 339
- drm_mode_plane_set_obj_prop (C function), 312
- drm_mode_probed_add (C function), 329
- drm_mode_property_enum (C struct), 787
- drm_mode_prune_invalid (C function), 339
- drm_mode_put_tile_group (C function), 385
- drm_mode_rect (C struct), 795
- DRM_MODE_RES_MM (C macro), 323
- drm_mode_revoke_lease (C struct), 795
- drm_mode_set (C struct), 273
- drm_mode_set_config_internal (C function), 278
- drm_mode_set_crtcinfo (C function), 335
- drm_mode_set_name (C function), 334
- drm_mode_sort (C function), 339
- drm_mode_status (C enum), 321
- drm_mode_validate_driver (C function), 337
- drm_mode_validate_size (C function), 338
- drm_mode_validate_ycbcr420 (C function), 338
- drm_mode_vrefresh (C function), 334
- drm_modeset_acquire_ctx (C struct), 400
- drm_modeset_acquire_fini (C function), 403
- drm_modeset_acquire_init (C function), 403
- drm_modeset_backoff (C function), 404
- drm_modeset_drop_locks (C function), 403
- drm_modeset_is_locked (C function), 401
- drm_modeset_lock (C function), 404
- drm_modeset_lock (C struct), 400
- drm_modeset_lock_all (C function), 402
- DRM_MODESET_LOCK_ALL_BEGIN (C macro), 401
- drm_modeset_lock_all_ctx (C function), 405
- DRM_MODESET_LOCK_ALL_END (C macro), 402
- drm_modeset_lock_assert_held (C function), 401
- drm_modeset_lock_fini (C function), 401
- drm_modeset_lock_init (C function), 404
- drm_modeset_lock_single_interruptible (C function), 404
- drm_modeset_unlock (C function), 405
- drm_modeset_unlock_all (C function), 402
- drm_monitor_range_info (C struct), 346
- drm_noop (C function), 765
- drm_nouveau_exec (C struct), 837
- drm_nouveau_exec_push (C struct), 837
- drm_nouveau_sync (C struct), 834
- drm_nouveau_vm_bind (C struct), 836
- drm_nouveau_vm_bind_op (C struct), 835
- drm_nouveau_vm_init (C struct), 835

`drm_object_attach_property` (C function), 224
`drm_object_properties` (C struct), 223
`drm_object_property_get_default_value` (C function), 226
`drm_object_property_get_value` (C function), 225
`drm_object_property_set_value` (C function), 225
`drm_of_component_match_add` (C function), 735
`drm_of_component_probe` (C function), 735
`drm_of_crtc_port_mask` (C function), 734
`drm_of_find_panel_or_bridge` (C function), 736
`drm_of_find_possible_crtcs` (C function), 735
`drm_of_get_data_lanes_count` (C function), 737
`drm_of_get_data_lanes_count_ep` (C function), 737
`drm_of_get_dsi_bus` (C function), 738
`drm_of_lvds_get_data_mapping` (C function), 737
`drm_of_lvds_get_dual_link_pixel_order` (C function), 736
`drm_open` (C function), 37
`drm_panel` (C struct), 579
`drm_panel_add` (C function), 580
`drm_panel_add_follower` (C function), 583
`drm_panel_bridge_add` (C function), 574
`drm_panel_bridge_add_typed` (C function), 574
`drm_panel_bridge_connector` (C function), 576
`drm_panel_bridge_remove` (C function), 574
`drm_panel_bridge_set_orientation` (C function), 575
`drm_panel_disable` (C function), 581
`drm_panel_dp_aux_backlight` (C function), 617
`drm_panel_enable` (C function), 581
`drm_panel_funcs` (C struct), 577
`drm_panel_get_modes` (C function), 581
`drm_panel_init` (C function), 580
`drm_panel_of_backlight` (C function), 584
`drm_panel_orientation` (C enum), 346
`drm_panel_prepare` (C function), 580
`drm_panel_remove` (C function), 580
`drm_panel_remove_follower` (C function), 583
`drm_panel_unprepare` (C function), 581
`drm_pending_event` (C struct), 32
`drm_pending_vblank_event` (C struct), 443
`drm_plane` (C struct), 304
`drm_plane_cleanup` (C function), 311
`drm_plane_create_alpha_property` (C function), 314
`drm_plane_create_blend_mode_property` (C function), 317
`drm_plane_create_color_properties` (C function), 280
`drm_plane_create_rotation_property` (C function), 314
`drm_plane_create_scaling_filter_property` (C function), 313
`drm_plane_create_zpos_immutable_property` (C function), 316
`drm_plane_create_zpos_property` (C function), 315
`drm_plane_enable_fb_damage_clips` (C function), 313
`drm_plane_find` (C function), 309
`drm_plane_force_disable` (C function), 312
`drm_plane_from_index` (C function), 311
`drm_plane_funcs` (C struct), 300
`drm_plane_get_damage_clips` (C function), 313
`drm_plane_get_damage_clips_count` (C function), 313
`drm_plane_helper_add` (C function), 479
`drm_plane_helper_atomic_check` (C function), 740
`drm_plane_helper_destroy` (C function), 740
`drm_plane_helper_disable_primary` (C function), 740
`drm_plane_helper_funcs` (C struct), 475
`drm_plane_helper_update_primary` (C function), 739
`drm_plane_index` (C function), 309
`drm_plane_mask` (C function), 309
`drm_plane_state` (C struct), 298
`drm_plane_type` (C enum), 304
`drm_poll` (C function), 39
`DRM_PRIME_CAP_EXPORT` (C macro), 776
`DRM_PRIME_CAP_IMPORT` (C macro), 776
`drm_prime_file_private` (C struct), 124
`drm_prime_gem_destroy` (C function), 130
`drm_prime_get_contiguous_size` (C function), 128
`drm_prime_pages_to_sg` (C function), 128
`drm_prime_sg_to_dma_addr_array` (C func-

- tion*), 130
- `drm_prime_sg_to_page_array` (C function), 130
- `drm_print_bits` (C function), 49
- `drm_print_iterator` (C struct), 44
- `drm_print_memory_stats` (C function), 42
- `drm_print_regset32` (C function), 49
- `drm_printer` (C struct), 43
- `drm_printf` (C function), 49
- `drm_printf_indent` (C macro), 44
- `drm_privacy_screen` (C struct), 746
- `drm_privacy_screen_call_notifier_chain` (C function), 750
- `drm_privacy_screen_get` (C function), 748
- `drm_privacy_screen_get_state` (C function), 749
- `drm_privacy_screen_lookup` (C struct), 747
- `drm_privacy_screen_lookup_add` (C function), 747
- `drm_privacy_screen_lookup_remove` (C function), 747
- `drm_privacy_screen_ops` (C struct), 745
- `drm_privacy_screen_put` (C function), 748
- `drm_privacy_screen_register` (C function), 750
- `drm_privacy_screen_register_notifier` (C function), 749
- `drm_privacy_screen_set_sw_state` (C function), 748
- `drm_privacy_screen_status` (C enum), 347
- `drm_privacy_screen_unregister` (C function), 750
- `drm_privacy_screen_unregister_notifier` (C function), 749
- `drm_private_obj` (C struct), 231
- `drm_private_state` (C struct), 232
- `drm_private_state_funcs` (C struct), 230
- `drm_probe_ddc` (C function), 701
- `drm_property` (C struct), 407
- `drm_property_add_enum` (C function), 414
- `drm_property_blob` (C struct), 409
- `drm_property_blob_get` (C function), 416
- `drm_property_blob_put` (C function), 415
- `drm_property_create` (C function), 410
- `drm_property_create_bitmask` (C function), 411
- `drm_property_create_blob` (C function), 415
- `drm_property_create_bool` (C function), 414
- `drm_property_create_enum` (C function), 411
- `drm_property_create_object` (C function), 413
- `drm_property_create_range` (C function), 412
- `drm_property_create_signed_range` (C function), 413
- `drm_property_destroy` (C function), 415
- `drm_property_enum` (C struct), 406
- `drm_property_find` (C function), 410
- `drm_property_lookup_blob` (C function), 416
- `drm_property_replace_blob` (C function), 417
- `drm_property_replace_global_blob` (C function), 416
- `drm_property_type_is` (C function), 410
- `drm_put_dev` (C function), 23
- `drm_puts` (C function), 48
- `drm_read` (C function), 38
- `drm_rect` (C struct), 723
- `drm_rect_adjust_size` (C function), 725
- `DRM_RECT_ARG` (C macro), 724
- `drm_rect_calc_hscale` (C function), 727
- `drm_rect_calc_vscale` (C function), 728
- `drm_rect_clip_scaled` (C function), 727
- `drm_rect_debug_print` (C function), 728
- `drm_rect_downscale` (C function), 725
- `drm_rect_equals` (C function), 726
- `DRM_RECT_FMT` (C macro), 724
- `DRM_RECT_FP_ARG` (C macro), 724
- `DRM_RECT_FP_FMT` (C macro), 724
- `drm_rect_fp_to_int` (C function), 727
- `drm_rect_height` (C function), 726
- `drm_rect_init` (C function), 724
- `DRM_RECT_INIT` (C macro), 723
- `drm_rect_intersect` (C function), 727
- `drm_rect_rotate` (C function), 729
- `drm_rect_rotate_inv` (C function), 729
- `drm_rect_translate` (C function), 725
- `drm_rect_translate_to` (C function), 725
- `drm_rect_visible` (C function), 726
- `drm_rect_width` (C function), 726
- `drm_release` (C function), 37
- `drm_release_noglobal` (C function), 38
- `drm_rotation_simplify` (C function), 315
- `drm_scdc_get_scrambling_status` (C function), 712
- `drm_scdc_read` (C function), 711
- `drm_scdc_readb` (C function), 710
- `drm_scdc_set_high_tmids_clock_ratio` (C function), 712
- `drm_scdc_set_scrambling` (C function), 712
- `drm_scdc_write` (C function), 711
- `drm_scdc_writeb` (C function), 711

`drm_sched_backend_ops` (C struct), 192
`drm_sched_entity` (C struct), 187
`drm_sched_entity_destroy` (C function), 202
`drm_sched_entity_error` (C function), 201
`drm_sched_entity_fini` (C function), 202
`drm_sched_entity_flush` (C function), 202
`drm_sched_entity_init` (C function), 201
`drm_sched_entity_modify_sched` (C function), 201
`drm_sched_entity_push_job` (C function), 203
`drm_sched_entity_set_priority` (C function), 202
`drm_sched_fault` (C function), 195
`drm_sched_fence` (C struct), 190
`DRM_SCHED_FENCE_DONT_PIPELINE` (C macro), 187
`DRM_SCHED_FENCE_FLAG_HAS_DEADLINE_BIT` (C macro), 187
`drm_sched_fini` (C function), 200
`drm_sched_increase_karma` (C function), 200
`drm_sched_init` (C function), 199
`drm_sched_job` (C struct), 191
`drm_sched_job_add_dependency` (C function), 197
`drm_sched_job_add_implicit_dependencies` (C function), 198
`drm_sched_job_add_resv_dependencies` (C function), 198
`drm_sched_job_add_syncobj_dependency` (C function), 198
`drm_sched_job_arm` (C function), 197
`drm_sched_job_cleanup` (C function), 199
`drm_sched_job_init` (C function), 197
`drm_sched_pick_best` (C function), 199
`drm_sched_resubmit_jobs` (C function), 196
`drm_sched_resume_timeout` (C function), 195
`drm_sched_rq` (C struct), 189
`drm_sched_start` (C function), 196
`drm_sched_stop` (C function), 196
`drm_sched_suspend_timeout` (C function), 195
`drm_scrambling` (C struct), 343
`drm_self_refresh_helper_alter_state` (C function), 586
`drm_self_refresh_helper_cleanup` (C function), 586
`drm_self_refresh_helper_init` (C function), 586
`drm_self_refresh_helper_update_avg_times` (C function), 585
`drm_send_event` (C function), 41
`drm_send_event_locked` (C function), 41
`drm_send_event_timestamp_locked` (C function), 41
`drm_seq_file_printer` (C function), 45
`drm_set_preferred_mode` (C function), 709
`DRM_SHADOW_PLANE_MAX_HEIGHT` (C macro), 512
`DRM_SHADOW_PLANE_MAX_WIDTH` (C macro), 511
`drm_shadow_plane_state` (C struct), 512
`drm_show_fdinfo` (C function), 42
`drm_show_memory_stats` (C function), 42
`drm_simple_display_pipe` (C struct), 520
`drm_simple_display_pipe_attach_bridge` (C function), 522
`drm_simple_display_pipe_funcs` (C struct), 518
`drm_simple_display_pipe_init` (C function), 522
`drm_simple_encoder_init` (C function), 522
`DRM_SIMPLE_MODE` (C macro), 323
`drm_state_dump` (C function), 256
`drm_syncobj` (C struct), 177
`drm_syncobj_add_point` (C function), 179
`drm_syncobj_create` (C function), 180
`drm_syncobj_eventfd` (C struct), 779
`drm_syncobj_fence_get` (C function), 178
`drm_syncobj_find` (C function), 178
`drm_syncobj_find_fence` (C function), 179
`drm_syncobj_free` (C function), 180
`drm_syncobj_get` (C function), 178
`drm_syncobj_get_fd` (C function), 181
`drm_syncobj_get_handle` (C function), 180
`drm_syncobj_put` (C function), 178
`drm_syncobj_replace_fence` (C function), 179
`drm_sysfs_connector_hotplug_event` (C function), 773
`drm_sysfs_connector_property_event` (C function), 774
`drm_sysfs_hotplug_event` (C function), 773
`drm_tile_group` (C struct), 369
`drm_timeout_abs_to_jiffies` (C function), 181
`drm_tv_connector_state` (C struct), 353
`drm_universal_plane_alloc` (C macro), 308
`drm_universal_plane_init` (C function), 310
`drm_vblank_crtc` (C struct), 444
`drm_vblank_init` (C function), 446
`drm_vblank_work` (C struct), 455
`drm_vblank_work_cancel_sync` (C function),

- 457
 drm_vblank_work_flush (C function), 457
 drm_vblank_work_init (C function), 457
 drm_vblank_work_schedule (C function), 456
 drm_vma_node_allow (C function), 121
 drm_vma_node_allow_once (C function), 121
 drm_vma_node_is_allowed (C function), 122
 drm_vma_node_offset_addr (C function), 118
 drm_vma_node_reset (C function), 117
 drm_vma_node_revoke (C function), 122
 drm_vma_node_size (C function), 117
 drm_vma_node_start (C function), 117
 drm_vma_node_unmap (C function), 118
 drm_vma_node_verify_access (C function), 118
 drm_vma_offset_add (C function), 120
 drm_vma_offset_exact_lookup_locked (C function), 116
 drm_vma_offset_lock_lookup (C function), 116
 drm_vma_offset_lookup_locked (C function), 119
 drm_vma_offset_manager_destroy (C function), 119
 drm_vma_offset_manager_init (C function), 119
 drm_vma_offset_remove (C function), 121
 drm_vma_offset_unlock_lookup (C function), 117
 drm_vprintf (C function), 43
 drm_vram_helper_mode_valid (C function), 113
 drm_vram_mm (C struct), 108
 drm_vram_mm_debugfs_init (C function), 113
 drm_vram_mm_of_bdev (C function), 108
 drm_wait_one_vblank (C function), 452
 drm_warn_on_modeset_not_all_locked (C function), 403
 drm_writeback_connector (C struct), 387
 drm_writeback_connector_init (C function), 389
 drm_writeback_connector_init_with_encoder (C function), 390
 drm_writeback_job (C struct), 388
 drm_writeback_queue_job (C function), 391
 drm_writeback_signal_completion (C function), 391
 drmm_add_action (C macro), 28
 drmm_add_action_or_reset (C macro), 28
 drmm_connector_init (C function), 372
 drmm_crtc_alloc_with_planes (C macro), 274
 drmm_crtc_init_with_planes (C function), 277
 drmm_encoder_alloc (C macro), 394
 drmm_encoder_init (C function), 398
 drmm_kcalloc (C function), 29
 drmm_kfree (C function), 27
 drmm_kmalloc (C function), 27
 drmm_kmalloc_array (C function), 29
 drmm_kstrdup (C function), 27
 drmm_kzalloc (C function), 28
 drmm_mode_config_init (C function), 221
 drmm_mutex_init (C macro), 29
 drmm_of_get_bridge (C function), 577
 drmm_panel_bridge_add (C function), 576
 drmm_plain_encoder_alloc (C macro), 395
 drmm_simple_encoder_alloc (C macro), 521
 drmm_universal_plane_alloc (C macro), 307
 drmm_vram_helper_init (C function), 113
 DRR, 941
 DSC, 940
 DWB, 941
 E
 ECP, 961
 EOP, 961
 F
 FB, 941
 FBC, 941
 FEC, 941
 for_each_if (C macro), 50
 for_each_new_connector_in_state (C macro), 239
 for_each_new_crtc_in_state (C macro), 240
 for_each_new_mst_mgr_in_state (C macro), 636
 for_each_new_plane_in_state (C macro), 243
 for_each_new_plane_in_state_reverse (C macro), 242
 for_each_new_private_obj_in_state (C macro), 244
 for_each_old_connector_in_state (C macro), 239
 for_each_old_crtc_in_state (C macro), 240
 for_each_old_mst_mgr_in_state (C macro), 636
 for_each_old_plane_in_state (C macro), 242
 for_each_old_private_obj_in_state (C macro), 243

`for_each_oldnew_connector_in_state` (C macro), 238
`for_each_oldnew_crtc_in_state` (C macro), 240
`for_each_oldnew_mst_mgr_in_state` (C macro), 635
`for_each_oldnew_plane_in_state` (C macro), 241
`for_each_oldnew_plane_in_state_reverse` (C macro), 241
`for_each_oldnew_private_obj_in_state` (C macro), 243
FRL, 941
`frontbuffer_flush` (C function), 975

G

GART, 961
GC, 961
GC0, 941
`gen7_append_oa_reports` (C function), 1087
`gen7_oa_read` (C function), 1088
`gen8_append_oa_reports` (C function), 1086
`gen8_oa_read` (C function), 1087
GMC, 961
GPUVM, 961
GSL, 941
GTT, 961

H

`hdmi_audio_infoframe_check` (C function), 717
`hdmi_audio_infoframe_init` (C function), 717
`hdmi_audio_infoframe_pack` (C function), 717
`hdmi_audio_infoframe_pack_for_dp` (C function), 718
`hdmi_audio_infoframe_pack_only` (C function), 717
`hdmi_avi_infoframe_check` (C function), 714
`hdmi_avi_infoframe_init` (C function), 714
`hdmi_avi_infoframe_pack` (C function), 715
`hdmi_avi_infoframe_pack_only` (C function), 715
`hdmi_drm_infoframe_check` (C function), 720
`hdmi_drm_infoframe_init` (C function), 719
`hdmi_drm_infoframe_pack` (C function), 720
`hdmi_drm_infoframe_pack_only` (C function), 720
`hdmi_drm_infoframe_unpack_only` (C function), 722
`hdmi_infoframe` (C union), 713
`hdmi_infoframe_check` (C function), 721
`hdmi_infoframe_log` (C function), 722
`hdmi_infoframe_pack` (C function), 721
`hdmi_infoframe_pack_only` (C function), 721
`hdmi_infoframe_unpack` (C function), 722
`hdmi_spd_infoframe_check` (C function), 716
`hdmi_spd_infoframe_init` (C function), 715
`hdmi_spd_infoframe_pack` (C function), 716
`hdmi_spd_infoframe_pack_only` (C function), 716
`hdmi_vendor_infoframe_check` (C function), 718
`hdmi_vendor_infoframe_init` (C function), 718
`hdmi_vendor_infoframe_pack` (C function), 719
`hdmi_vendor_infoframe_pack_only` (C function), 719
`hdr_metadata_infoframe` (C struct), 790
`hdr_output_metadata` (C struct), 791
`hdr_sink_metadata` (C struct), 713
`hostlx_bo_cache` (C struct), 1098
`hostlx_client` (C struct), 1099
`hostlx_client_exit` (C function), 1101
`hostlx_client_ops` (C struct), 1098
`hostlx_client_unregister` (C function), 1102
`hostlx_device_exit` (C function), 1101
`hostlx_device_init` (C function), 1100
`hostlx_driver` (C struct), 1100
`hostlx_driver_register_full` (C function), 1101
`hostlx_driver_unregister` (C function), 1101
`hostlx_syncpt_alloc` (C function), 1102
`hostlx_syncpt_base_id` (C function), 1105
`hostlx_syncpt_get` (C function), 1105
`hostlx_syncpt_get_base` (C function), 1105
`hostlx_syncpt_get_by_id` (C function), 1104
`hostlx_syncpt_get_by_id_noref` (C function), 1105
`hostlx_syncpt_id` (C function), 1102
`hostlx_syncpt_incr` (C function), 1103
`hostlx_syncpt_incr_max` (C function), 1103
`hostlx_syncpt_put` (C function), 1104
`hostlx_syncpt_read` (C function), 1104
`hostlx_syncpt_read_max` (C function), 1104
`hostlx_syncpt_read_min` (C function), 1104
`hostlx_syncpt_release_vblank_reservation` (C function), 1105
`hostlx_syncpt_request` (C function), 1103

- [hostlx_syncpt_wait \(C function\), 1103](#)
[hpd_rx_irq_offload_work \(C struct\), 903](#)
[hpd_rx_irq_offload_work_queue \(C struct\), 902](#)
[HQD, 962](#)
I
[i915_audio_component \(C struct\), 986](#)
[i915_audio_component_cleanup \(C function\), 986](#)
[i915_audio_component_init \(C function\), 985](#)
[i915_cmd_parser_get_version \(C function\), 1025](#)
[i915_context_engines_parallel_submit \(C struct\), 813](#)
[i915_engine_class_instance \(C struct\), 803](#)
[i915_gem_context \(C struct\), 1030](#)
[i915_gem_engine_type \(C enum\), 1027](#)
[i915_gem_engines \(C struct\), 1026](#)
[i915_gem_engines_iter \(C struct\), 1026](#)
[i915_gem_evict_for_node \(C function\), 1020](#)
[i915_gem_evict_something \(C function\), 1020](#)
[i915_gem_evict_vm \(C function\), 1021](#)
[i915_gem_fence_alignment \(C function\), 1044](#)
[i915_gem_fence_size \(C function\), 1044](#)
[i915_gem_get_tiling_ioctl \(C function\), 1045](#)
[i915_gem_gtt_insert \(C function\), 1040](#)
[i915_gem_gtt_reserve \(C function\), 1039](#)
[i915_gem_object_do_bit_17_swizzle \(C function\), 1042](#)
[i915_gem_object_make_purgeable \(C function\), 1023](#)
[i915_gem_object_make_shrinkable \(C function\), 1023](#)
[i915_gem_object_make_unshrinkable \(C function\), 1022](#)
[i915_gem_object_save_bit_17_swizzle \(C function\), 1042](#)
[i915_gem_proto_context \(C struct\), 1028](#)
[i915_gem_proto_engine \(C struct\), 1027](#)
[i915_gem_set_tiling_ioctl \(C function\), 1045](#)
[i915_gem_shrink \(C function\), 1021](#)
[i915_gem_shrink_all \(C function\), 1022](#)
[i915_oa_ops \(C struct\), 1081](#)
[i915_oa_poll_wait \(C function\), 1083](#)
[i915_oa_read \(C function\), 1082](#)
[i915_oa_stream_disable \(C function\), 1083](#)
[i915_oa_stream_enable \(C function\), 1082](#)
[i915_oa_stream_init \(C function\), 1081](#)
[i915_oa_wait_unlocked \(C function\), 1083](#)
[i915_perf_add_config_ioctl \(C function\), 1073](#)
[i915_perf_destroy_locked \(C function\), 1078](#)
[i915_perf_disable_locked \(C function\), 1079](#)
[i915_perf_enable_locked \(C function\), 1079](#)
[i915_perf_fini \(C function\), 1071](#)
[i915_perf_init \(C function\), 1071](#)
[i915_perf_ioctl \(C function\), 1079](#)
[i915_perf_ioctl_locked \(C function\), 1089](#)
[i915_perf_ioctl_version \(C function\), 1089](#)
[i915_perf_open_ioctl \(C function\), 1072](#)
[i915_perf_open_ioctl_locked \(C function\), 1077](#)
[i915_perf_poll \(C function\), 1080](#)
[i915_perf_poll_locked \(C function\), 1080](#)
[i915_perf_read \(C function\), 1078](#)
[i915_perf_register \(C function\), 1071](#)
[i915_perf_release \(C function\), 1072](#)
[i915_perf_remove_config_ioctl \(C function\), 1073](#)
[i915_perf_stream \(C struct\), 1074](#)
[i915_perf_stream_ops \(C struct\), 1076](#)
[i915_perf_unregister \(C function\), 1072](#)
[i915_reserve_fence \(C function\), 1041](#)
[i915_sched_engine \(C struct\), 1035](#)
[i915_unreserve_fence \(C function\), 1041](#)
[i915_user_extension \(C struct\), 801](#)
[i915_vma_pin_fence \(C function\), 1041](#)
[i915_vma_revoke_fence \(C function\), 1041](#)
[IB, 962](#)
[icl_set_active_port_dppll \(C function\), 1005](#)
[iGPU, 941](#)
[IH, 961](#)
[intel_audio_codec_disable \(C function\), 985](#)
[intel_audio_codec_enable \(C function\), 985](#)
[intel_audio_deinit \(C function\), 986](#)
[intel_audio_hooks_init \(C function\), 985](#)
[intel_audio_init \(C function\), 986](#)
[intel_bios_driver_remove \(C function\), 997](#)
[intel_bios_init \(C function\), 997](#)
[intel_bios_is_dsi_present \(C function\), 998](#)
[intel_bios_is_lvds_present \(C function\),](#)

[998](#)
[intel_bios_is_port_present \(C function\), 998](#)
[intel_bios_is_tv_present \(C function\), 998](#)
[intel_bios_is_valid_vbt \(C function\), 997](#)
[intel_cdclk_can_cd2x_update \(C function\), 1001](#)
[intel_cdclk_changed \(C function\), 1001](#)
[intel_cdclk_init_hw \(C function\), 1000](#)
[intel_cdclk_needs_modeset \(C function\), 1000](#)
[intel_cdclk_uninit_hw \(C function\), 1000](#)
[intel_check_cpu_fifo_underruns \(C function\), 979](#)
[intel_check_pch_fifo_underruns \(C function\), 979](#)
[intel_cleanup_plane_fb \(C function\), 980](#)
[intel_compute_shared_dplls \(C function\), 1005](#)
[intel_cpu_fifo_underrun_irq_handler \(C function\), 978](#)
[intel_disable_shared_dpll \(C function\), 1004](#)
[intel_dmc_disable_program \(C function\), 996](#)
[intel_dmc_fini \(C function\), 996](#)
[intel_dmc_init \(C function\), 996](#)
[intel_dmc_load_program \(C function\), 995](#)
[intel_dmc_resume \(C function\), 996](#)
[intel_dmc_suspend \(C function\), 996](#)
[intel_dppll_dump_hw_state \(C function\), 1007](#)
[intel_dppll_get_freq \(C function\), 1007](#)
[intel_dppll_get_hw_state \(C function\), 1007](#)
[intel_dppll_id \(C enum\), 1008](#)
[intel_drrs_activate \(C function\), 993](#)
[intel_drrs_crtc_init \(C function\), 994](#)
[intel_drrs_deactivate \(C function\), 993](#)
[intel_drrs_flush \(C function\), 993](#)
[intel_drrs_invalidate \(C function\), 993](#)
[intel_dsb_cleanup \(C function\), 1012](#)
[intel_dsb_commit \(C function\), 1011](#)
[intel_dsb_prepare \(C function\), 1011](#)
[intel_dsb_reg_write \(C function\), 1011](#)
[intel_enable_shared_dpll \(C function\), 1003](#)
[intel_engine_cleanup_cmd_parser \(C function\), 1025](#)
[intel_engine_cmd_parser \(C function\), 1025](#)
[intel_engine_init_cmd_parser \(C function\), 1024](#)
[intel_fbc_disable \(C function\), 991](#)
[intel_fbc_handle_fifo_underrun_irq \(C function\), 991](#)
[intel_fbc_init \(C function\), 992](#)
[intel_fbc_sanitise \(C function\), 992](#)
[intel_frontbuffer_flip \(C function\), 976](#)
[intel_frontbuffer_flip_complete \(C function\), 976](#)
[intel_frontbuffer_flip_prepare \(C function\), 976](#)
[intel_frontbuffer_flush \(C function\), 975](#)
[intel_frontbuffer_invalidate \(C function\), 975](#)
[intel_frontbuffer_track \(C function\), 977](#)
[intel_get_shared_dppll_by_id \(C function\), 1003](#)
[intel_gggtt_restore_fences \(C function\), 1042](#)
[intel_gt_mcr_get_nonterminated_steering \(C function\), 1015](#)
[intel_gt_mcr_get_ss_steering \(C function\), 1016](#)
[intel_gt_mcr_lock \(C function\), 1012](#)
[intel_gt_mcr_multicast_rmw \(C function\), 1015](#)
[intel_gt_mcr_multicast_write \(C function\), 1014](#)
[intel_gt_mcr_multicast_write_fw \(C function\), 1014](#)
[intel_gt_mcr_read \(C function\), 1013](#)
[intel_gt_mcr_read_any \(C function\), 1016](#)
[intel_gt_mcr_read_any_fw \(C function\), 1015](#)
[intel_gt_mcr_unicast_write \(C function\), 1013](#)
[intel_gt_mcr_unlock \(C function\), 1013](#)
[intel_gt_mcr_wait_for_reg \(C function\), 1017](#)
[intel_guc \(C struct\), 1050](#)
[intel_guc_allocate_vma \(C function\), 1055](#)
[intel_guc_fw_upload \(C function\), 1056](#)
[intel_guc_gggtt_offset \(C function\), 1053](#)
[intel_gvt_driver_remove \(C function\), 972](#)
[intel_gvt_init \(C function\), 972](#)
[intel_gvt_resume \(C function\), 973](#)
[intel_hpd_init \(C function\), 983](#)
[intel_hpd_irq_handler \(C function\), 983](#)
[intel_hpd_irq_storm_detect \(C function\), 982](#)
[intel_hpd_pin_default \(C function\), 982](#)
[intel_hpd_poll_disable \(C function\), 984](#)

- `intel_hpd_poll_enable` (C function), 983
- `intel_hpd_trigger_irq` (C function), 983
- `intel_huc_auth` (C function), 1067
- `intel_init_cdclk_hooks` (C function), 1003
- `intel_irq_init` (C function), 970
- `intel_lpe_audio_init` (C function), 987
- `intel_lpe_audio_irq_handler` (C function), 987
- `intel_lpe_audio_notify` (C function), 987
- `intel_lpe_audio_tear_down` (C function), 987
- `intel_pch_fifo_underrun_irq_handler` (C function), 978
- `intel_plane_destroy_state` (C function), 980
- `intel_plane_duplicate_state` (C function), 979
- `intel_prepare_plane_fb` (C function), 980
- `intel_psr_disable` (C function), 989
- `intel_psr_flush` (C function), 990
- `intel_psr_init` (C function), 990
- `intel_psr_invalidate` (C function), 989
- `intel_psr_lock` (C function), 990
- `intel_psr_pause` (C function), 989
- `intel_psr_resume` (C function), 989
- `intel_psr_unlock` (C function), 991
- `intel_psr_wait_for_idle_locked` (C function), 989
- `intel_pxp` (C struct), 1046
- `intel_read_rawclk` (C function), 1002
- `intel_reference_shared_dpll_crtc` (C function), 1004
- `intel_release_shared_dppls` (C function), 1006
- `intel_reserve_shared_dppls` (C function), 1006
- `intel_runtime_pm_disable_interrupts` (C function), 970
- `intel_runtime_pm_enable` (C function), 965
- `intel_runtime_pm_enable_interrupts` (C function), 970
- `intel_runtime_pm_get` (C function), 963
- `intel_runtime_pm_get_noresume` (C function), 964
- `intel_runtime_pm_get_raw` (C function), 963
- `intel_runtime_pm_put` (C function), 965
- `intel_runtime_pm_put_raw` (C function), 965
- `intel_runtime_pm_put_unchecked` (C function), 965
- `intel_set_cdclk` (C function), 1001
- `intel_set_cdclk_post_plane_update` (C function), 1002
- `intel_set_cdclk_pre_plane_update` (C function), 1002
- `intel_set_cpu_fifo_underrun_reporting` (C function), 977
- `intel_set_pch_fifo_underrun_reporting` (C function), 978
- `intel_shared_dppll` (C struct), 1010
- `intel_shared_dppll_init` (C function), 1005
- `intel_shared_dppll_state` (C struct), 1009
- `intel_shared_dppll_swap_state` (C function), 1004
- `intel_uncore_forcewake_flush` (C function), 967
- `intel_uncore_forcewake_for_reg` (C function), 969
- `intel_uncore_forcewake_get` (C function), 966
- `intel_uncore_forcewake_get__locked` (C function), 966
- `intel_uncore_forcewake_put` (C function), 967
- `intel_uncore_forcewake_put__locked` (C function), 967
- `intel_uncore_forcewake_user_get` (C function), 966
- `intel_uncore_forcewake_user_put` (C function), 966
- `intel_unreference_shared_dppll_crtc` (C function), 1004
- `intel_update_active_dppll` (C function), 1006
- `intel_update_cdclk` (C function), 1002
- `intel_update_max_cdclk` (C function), 1002
- `intel_vgpu_detect` (C function), 971
- `intel_vgt_balloon` (C function), 971
- `intel_vgt_deballoon` (C function), 971
- IP, 962
- ISR, 941
- ISV, 941
- K
- KCQ, 962
- KGQ, 962
- KIQ, 962
- KMD, 941
- `komeda_component` (C struct), 1125
- `komeda_component_output` (C struct), 1127
- `komeda_component_state` (C struct), 1127
- `komeda_crtc` (C struct), 1139
- `komeda_crtc_atomic_check` (C function), 1140

`komeda_crtc_state` (*C struct*), 1140
`komeda_dev` (*C struct*), 1132
`komeda_dev_funcs` (*C struct*), 1131
`komeda_fb` (*C struct*), 1135
`komeda_format_caps` (*C struct*), 1134
`komeda_format_caps_table` (*C struct*), 1135
`komeda_pipeline` (*C struct*), 1129
`komeda_pipeline_state` (*C struct*), 1130
`komeda_plane` (*C struct*), 1138
`komeda_plane_atomic_check` (*C function*), 1140
`komeda_plane_state` (*C struct*), 1138
`komeda_wb_connector` (*C struct*), 1139

L

LB, 941
LFC, 941
LTPR, 941
LUT, 941

M

MALL, 941
MC, 941
MEC, 962
MES, 962
`mipi_dbi` (*C struct*), 652
`mipi_dbi_buf_copy` (*C function*), 655
`mipi_dbi_command` (*C macro*), 654
`mipi_dbi_command_buf` (*C function*), 655
`mipi_dbi_command_read` (*C function*), 655
`mipi_dbi_debugfs_init` (*C function*), 662
`mipi_dbi_dev` (*C struct*), 653
`mipi_dbi_dev_init` (*C function*), 659
`mipi_dbi_dev_init_with_formats` (*C function*), 659
`mipi_dbi_display_is_on` (*C function*), 660
`mipi_dbi_enable_flush` (*C function*), 656
`mipi_dbi_hw_reset` (*C function*), 660
`mipi_dbi_pipe_begin_fb_access` (*C function*), 657
`mipi_dbi_pipe_destroy_plane_state` (*C function*), 658
`mipi_dbi_pipe_disable` (*C function*), 657
`mipi_dbi_pipe_duplicate_plane_state` (*C function*), 658
`mipi_dbi_pipe_end_fb_access` (*C function*), 657
`mipi_dbi_pipe_mode_valid` (*C function*), 656
`mipi_dbi_pipe_reset_plane` (*C function*), 658
`mipi_dbi_pipe_update` (*C function*), 656

`mipi_dbi_poweron_conditional_reset` (*C function*), 661
`mipi_dbi_poweron_reset` (*C function*), 660
`mipi_dbi_spi_cmd_max_speed` (*C function*), 661
`mipi_dbi_spi_init` (*C function*), 661
`mipi_dbi_spi_transfer` (*C function*), 662
`mipi_dsi_attach` (*C function*), 669
`mipi_dsi_compression_mode` (*C function*), 670
`mipi_dsi_create_packet` (*C function*), 670
`mipi_dsi_dcs_enter_sleep_mode` (*C function*), 674
`mipi_dsi_dcs_exit_sleep_mode` (*C function*), 674
`mipi_dsi_dcs_get_display_brightness` (*C function*), 676
`mipi_dsi_dcs_get_display_brightness_large` (*C function*), 677
`mipi_dsi_dcs_get_pixel_format` (*C function*), 674
`mipi_dsi_dcs_get_power_mode` (*C function*), 673
`mipi_dsi_dcs_nop` (*C function*), 673
`mipi_dsi_dcs_read` (*C function*), 673
`mipi_dsi_dcs_set_column_address` (*C function*), 675
`mipi_dsi_dcs_set_display_brightness` (*C function*), 676
`mipi_dsi_dcs_set_display_brightness_large` (*C function*), 677
`mipi_dsi_dcs_set_display_off` (*C function*), 674
`mipi_dsi_dcs_set_display_on` (*C function*), 675
`mipi_dsi_dcs_set_page_address` (*C function*), 675
`mipi_dsi_dcs_set_pixel_format` (*C function*), 676
`mipi_dsi_dcs_set_tear_off` (*C function*), 675
`mipi_dsi_dcs_set_tear_on` (*C function*), 675
`mipi_dsi_dcs_set_tear_scanline` (*C function*), 676
`mipi_dsi_dcs_soft_reset` (*C function*), 673
`mipi_dsi_dcs_tear_mode` (*C enum*), 666
`mipi_dsi_dcs_write` (*C function*), 672
`mipi_dsi_dcs_write_buffer` (*C function*), 672
`mipi_dsi_dcs_write_seq` (*C macro*), 667
`mipi_dsi_detach` (*C function*), 669

- mipi_dsi_device (*C struct*), 665
- mipi_dsi_device_info (*C struct*), 665
- mipi_dsi_device_register_full (*C function*), 668
- mipi_dsi_device_unregister (*C function*), 668
- mipi_dsi_driver (*C struct*), 667
- mipi_dsi_driver_register_full (*C function*), 677
- mipi_dsi_driver_unregister (*C function*), 677
- mipi_dsi_generic_read (*C function*), 671
- mipi_dsi_generic_write (*C function*), 671
- mipi_dsi_generic_write_seq (*C macro*), 667
- mipi_dsi_host (*C struct*), 664
- mipi_dsi_host_ops (*C struct*), 664
- mipi_dsi_msg (*C struct*), 663
- mipi_dsi_packet (*C struct*), 663
- mipi_dsi_packet_format_is_long (*C function*), 670
- mipi_dsi_packet_format_is_short (*C function*), 669
- mipi_dsi_picture_parameter_set (*C function*), 671
- mipi_dsi_pixel_format_to_bpp (*C function*), 666
- mipi_dsi_shutdown_peripheral (*C function*), 670
- mipi_dsi_turn_on_peripheral (*C function*), 670
- MMHUB, 962
- MPC/MPCC, 941
- mpc_color_caps (*C struct*), 920
- mpcc_alpha_blend_mode (*C enum*), 925
- mpcc_blnd_cfg (*C struct*), 922
- MPO, 941
- MQD, 962
- MST, 942
- N**
- NBIO, 942
- NBP State, 942
- O**
- oa_buffer_check_unlocked (*C function*), 1085
- oa_get_render_ctx_id (*C function*), 1088
- oa_put_render_ctx_id (*C function*), 1089
- ODM, 942
- of_drm_find_bridge (*C function*), 572
- of_drm_find_panel (*C function*), 582
- of_drm_get_panel_orientation (*C function*), 582
- of_find_backlight_by_node (*C function*), 1151
- of_find_mipi_dsi_device_by_node (*C function*), 668
- of_find_mipi_dsi_host_by_node (*C function*), 669
- of_get_drm_display_mode (*C function*), 333
- of_get_drm_panel_display_mode (*C function*), 334
- OPM, 942
- OPP, 942
- OPTC, 942
- OTG, 942
- P**
- PCON, 942
- perf_open_properties (*C struct*), 1084
- PGFSM, 942
- pipe_split_policy (*C enum*), 921
- PPLib, 962
- PSP, 962
- PSR, 942
- R**
- read_properties_unlocked (*C function*), 1077
- rfc.i915_context_engines_parallel_submit (*C struct*), 1196
- RLC, 962
- rom_curve_caps (*C struct*), 918
- S**
- SCL, 942
- SDMA, 962
- SDP, 942
- SE, 962
- SH, 962
- SLS, 942
- SMU, 962
- SS, 962
- SST, 942
- switch_power_state (*C enum*), 12
- T**
- TMDS, 942
- TMZ, 942
- to_drm_shadow_plane_state (*C function*), 512
- to_drm_vblank_work (*C macro*), 456
- ttm_agp_tt_create (*C function*), 68

ttm_bus_placement (*C struct*), 57
ttm_caching (*C enum*), 53
ttm_device (*C struct*), 54
ttm_device_init (*C function*), 55
ttm_global (*C struct*), 54
ttm_kmap_iter_iomap (*C struct*), 59
ttm_kmap_iter_iomap_init (*C function*), 63
ttm_kmap_iter_linear_io (*C struct*), 60
ttm_kmap_iter_tt (*C struct*), 65
ttm_kmap_iter_tt_init (*C function*), 68
ttm_lru_bulk_move (*C struct*), 59
ttm_lru_bulk_move_init (*C function*), 62
ttm_lru_bulk_move_pos (*C struct*), 59
ttm_lru_bulk_move_tail (*C function*), 62
ttm_place (*C struct*), 56
ttm_placement (*C struct*), 56
ttm_pool (*C struct*), 69
ttm_pool_alloc (*C function*), 70
ttm_pool_debugfs (*C function*), 71
ttm_pool_fini (*C function*), 71
ttm_pool_free (*C function*), 70
ttm_pool_init (*C function*), 70
ttm_pool_type (*C struct*), 69
ttm_resource (*C struct*), 58
ttm_resource_cursor (*C struct*), 59
ttm_resource_fini (*C function*), 62
ttm_resource_init (*C function*), 62
ttm_resource_manager (*C struct*), 57
ttm_resource_manager_cleanup (*C function*),
61
ttm_resource_manager_create_debugfs (*C function*), 64
ttm_resource_manager_debug (*C function*),
63
ttm_resource_manager_for_each_res (*C macro*), 61
ttm_resource_manager_init (*C function*), 63
ttm_resource_manager_set_used (*C function*), 61
ttm_resource_manager_usage (*C function*),
63
ttm_resource_manager_used (*C function*), 61
ttm_tt (*C struct*), 64
ttm_tt_create (*C function*), 66
ttm_tt_destroy (*C function*), 67
ttm_tt_fini (*C function*), 67
ttm_tt_init (*C function*), 66
ttm_tt_mark_for_clear (*C function*), 68
ttm_tt_populate (*C function*), 67
ttm_tt_swapin (*C function*), 67
ttm_tt_unpopulate (*C function*), 67

TTU, 942

U

unregister_all_irq_handlers (*C function*),
910

UVD, 942

V

vblank_control_work (*C struct*), 900

vbt_header (*C struct*), 998

VCE, 962

VCN, 962

vga_client_register (*C function*), 1171

vga_default_device (*C function*), 1169

vga_get (*C function*), 1169

vga_get_interruptible (*C function*), 1168

vga_get_uninterruptible (*C function*), 1169

vga_put (*C function*), 1170

vga_remove_vgacon (*C function*), 1169

vga_set_legacy_decoding (*C function*), 1170

vga_switcheroo_client (*C struct*), 1162

vga_switcheroo_client_fb_set (*C function*),
1157

vga_switcheroo_client_id (*C enum*), 1161

vga_switcheroo_client_ops (*C struct*), 1160

vga_switcheroo_client_probe_defer (*C function*), 1156

vga_switcheroo_get_client_state (*C function*), 1157

vga_switcheroo_handler (*C struct*), 1159

vga_switcheroo_handler_flags (*C function*),
1155

vga_switcheroo_handler_flags_t (*C enum*),
1160

vga_switcheroo_init_domain_pm_ops (*C function*), 1158

vga_switcheroo_lock_ddc (*C function*), 1157

vga_switcheroo_process_delayed_switch
(*C function*), 1158

vga_switcheroo_register_audio_client (*C function*), 1156

vga_switcheroo_register_client (*C function*), 1156

vga_switcheroo_register_handler (*C function*), 1155

vga_switcheroo_state (*C enum*), 1161

vga_switcheroo_unlock_ddc (*C function*),
1158

vga_switcheroo_unregister_client (*C function*), 1157

vga_switcheroo_unregister_handler (*C function*), 1155

`vgasr_priv` (*C struct*), [1162](#)

VRR, [942](#)

VTG, [940](#)