Linux GPU Driver Developer's Guide

Release 4.13.0-rc4+

The kernel development community

Sep 05, 2017

CHAPTER

ONE

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 check 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 .

1.2 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.

CHAPTER

TWO

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

Driver Features

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

u32 driver_features;

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

DRIVER_LEGACY Denote a legacy driver using shadow attach. Don't use.

DRIVER_KMS_LEGACY_CONTEXT Used only by nouveau for backwards compatibility with existing userspace. Don't use.

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

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

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

DRIVER_HAVE_IRQ; DRIVER_IRQ_SHARED DRIVER_HAVE_IRQ indicates whether the driver has an IRQ handler managed by the DRM Core. The core will support simple IRQ handler installation when the flag is set. The installation process is described in ?.

DRIVER_IRQ_SHARED indicates whether the device & handler support shared IRQs (note that this is required of PCI drivers).

DRIVER_GEM Driver use the GEM memory manager.

DRIVER_MODESET Driver supports mode setting interfaces (KMS).

DRIVER_PRIME Driver implements DRM PRIME buffer sharing.

DRIVER_RENDER Driver supports dedicated render nodes.

DRIVER_ATOMIC Driver supports atomic properties. In this case the driver must implement appropriate obj->atomic_get_property() vfuncs for any modeset objects with driver specific properties.

DRIVER_SYNCOBJ Driver support drm sync objects.

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 Device Instance and Driver Handling

A device instance for a drm driver is represented by `struct drm_device`. This is allocated with *`drm_dev_alloc()`*, usually from bus-specific ->:c:func:*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 intial output configuration plus obviously initialize all the corresponding hardware bits. An important part of this is also calling *`drm_dev_set_unique()`* to set the userspace-visible unique name of this device instance. 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 initalizing 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_unref()*.

Note that the lifetime rules for drm_device instance has still a lot of historical baggage. Hence use the reference counting provided by *drm_dev_ref()* and *drm_dev_unref()* only carefully.

It is recommended that drivers embed struct drm_device into their own device structure, which is supported through *drm_dev_init()*.

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 *);
  u32 (* get_vblank_counter) (struct drm_device *dev, unsigned int pipe);
  int (* enable_vblank) (struct drm_device *dev, unsigned int pipe);
  void (* disable_vblank) (struct drm_device *dev, unsigned int pipe);
  bool (* get_scanout_position) (struct drm_device *dev, unsigned int pipe,bool in_vblank_irq,␣
↪int *vpos, int *hpos,ktime_t *stime, ktime_t *etime, const struct drm_display_mode *mode);
  bool (* get_vblank_timestamp) (struct drm_device *dev, unsigned int pipe,int *max_error,
↪struct timeval *vblank_time, bool in_vblank_irq);
  irqreturn_t(* irq_handler) (int irq, void *arg);
  void (* irq_preinstall) (struct drm_device *dev);
  int (* irq_postinstall) (struct drm_device *dev);
  void (* irq_uninstall) (struct drm_device *dev);
  int (* master_create) (struct drm_device *dev, struct drm_master *master);
  void (* master_destroy) (struct drm_device *dev, struct drm_master *master);
  int (* 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 (* gem_free_object) (struct drm_gem_object *obj);
  void (* gem_free_object_unlocked) (struct drm_gem_object *obj);
  struct drm_gem_object *(* gem_create_object) (struct drm_device *dev, size_t size);
  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);
  int (* dumb_destroy) (struct drm_file *file_priv,struct drm_device *dev, uint32_t handle);
};
```

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 *drm_dev_init()* 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/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 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 ownershipe 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_unref()* 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. Drivers using this callback are responsible for calling *drm_dev_fini()* to finalize the device and then freeing the struct themselves.

get_vblank_counter Driver callback for fetching a raw hardware vblank counter for the CRTC specified with the pipe argument. 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.

This is deprecated and should not be used by new drivers. Use *drm_crtc_funcs.get_vblank_counter* instead.

Returns:

Raw vblank counter value.

enable_vblank Enable vblank interrupts for the CRTC specified with the pipe argument.

This is deprecated and should not be used by new drivers. Use *drm_crtc_funcs.enable_vblank* instead.

Returns:

Zero on success, appropriate errno if the given **crtc**'s vblank interrupt cannot be enabled.

disable_vblank Disable vblank interrupts for the CRTC specified with the pipe argument.

This is deprecated and should not be used by new drivers. Use *drm_crtc_funcs.disable_vblank* instead.

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 position was measured. Note that this is a helper callback which is only used if a driver uses *drm_calc_vbltimestamp_from_scanoutpos()* for the **get_vblank_timestamp** callback.

Parameters:

dev: DRM device.

pipe: Id of the crtc to query.

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.

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.

FIXME:

Since this is a helper to implement **get_vblank_timestamp**, we should move it to *struct drm_crtc_helper_funcs*, like all the other helper-internal hooks.

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.

Paramters:

dev: dev DRM device handle.

pipe: 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()*.

FIXME:

We should move this hook to *struct drm_crtc_funcs* like all the other vblank hooks.

irq_handler Interrupt handler called when using *drm_irq_install()*. Not used by drivers which implement their own interrupt handling.

irq_preinstall Optional callback used by *drm_irq_install()* which is called before the interrupt handler is registered. This should be used to clear out any pending interrupts (from e.g. firmware based drives) and reset the interrupt handling registers.

irq_postinstall Optional callback used by *drm_irq_install()* which is called after the interrupt handler is registered. This should be used to enable interrupt generation in the hardware.

irq_uninstall Optional callback used by *drm_irq_uninstall()* which is called before the interrupt handler is unregistered. This should be used to disable interrupt generation in the hardware.

master_create Called whenever a new master is created. Only used by vmwgfx.

master_destroy Called whenever a master is destroyed. Only used by vmwgfx.

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.

gem_free_object deconstructor for drm_gem_objects

This is deprecated and should not be used by new drivers. Use **gem_free_object_unlocked** instead.

gem_free_object_unlocked deconstructor for drm_gem_objects

This is for drivers which are not encumbered with `drm_device.struct_mutex` legacy locking schemes. Use this hook instead of **gem_free_object**.

gem_create_object constructor for gem objects

Hook for allocating the GEM object struct, for use by core helpers.

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. GEM-based drivers must use *drm_gem_create_mmap_offset()* to implement this.

Called by the user via ioctl.

Returns:

Zero on success, negative errno on failure.

dumb_destroy This destroys the userspace handle for the given dumb backing storage buffer. Since buffer objects must be reference counted in the kernel a buffer object won't be immediately freed if a framebuffer modeset object still uses it.

Called by the user via ioctl.

Returns:

Zero on success, negative errno on failure.

Description

This structure represent the common code for a family of cards. There will one 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.

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_unref()* 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.

int **drm_dev_init**(struct drm_device * *dev*, struct *drm_driver* * *driver*, struct device * *parent*)
Initialise new DRM device

Parameters

struct drm_device * dev DRM device

struct drm_driver * driver DRM driver

struct device * parent Parent device object

Description

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 subsystems. 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_ref()* and *drm_dev_unref()* to take and drop further ref-counts.

Note that for purely virtual devices **parent** can be NULL.

Drivers that do not want to allocate their own device struct embedding `struct drm_device` can call *drm_dev_alloc()* instead. For drivers that do embed `struct drm_device` it must be placed first in the overall structure, and the overall structure must be allocated using `kmalloc()`: The drm core's release function unconditionally calls `kfree()` on the **dev** pointer when the final reference is released. To override this behaviour, and so allow embedding of the drm_device inside the driver's device struct at an arbitrary offset, you must supply a *drm_driver.release* callback and control the finalization explicitly.

Return

0 on success, or error code on failure.

void **drm_dev_fini**(struct drm_device * *dev*)
Finalize a dead DRM device

Parameters

struct drm_device * dev DRM device

Description

Finalize a dead DRM device. This is the converse to *drm_dev_init()* and frees up all data allocated by it. All driver private data should be finalized first. Note that this function does not free the **dev**, that is left to the caller.

The ref-count of **dev** must be zero, and *drm_dev_fini()* should only be called from a *drm_driver.release* callback.

struct drm_device * **drm_dev_alloc**(struct *drm_driver* * *driver*, struct device * *parent*)
Allocate new DRM device

Parameters

struct drm_driver * driver DRM driver to allocate device for

struct device * parent Parent device object

Description

Allocate 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 subsystems. 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_ref()* and *drm_dev_unref()* to take and drop further ref-counts.

Note that for purely virtual devices **parent** can be NULL.

Drivers that wish to subclass or embed `struct drm_device` into their own struct should look at using *drm_dev_init()* instead.

Return

Pointer to new DRM device, or ERR_PTR on failure.

void **drm_dev_ref**(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_unref()* 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_unref**(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 .:c:func:*load()* function

Description

Register the DRM device **dev** with the system, advertise device to user-space and start normal device operation. **dev** must be allocated via *drm_dev_alloc()* 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_unref()* to drop their final reference.

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

int **drm_dev_set_unique**(struct drm_device * *dev*, const char * *name*)
Set the unique name of a DRM device

Parameters

struct drm_device * dev device of which to set the unique name

const char * name unique name

Description

Sets the unique name of a DRM device using the specified string. Drivers can use this at driver probe time if the unique name of the devices they drive is static.

Return

0 on success or a negative error code on failure.

2.1.3 Driver Load

IRQ Helper Library

The DRM core provides very simple support helpers to enable IRQ handling on a device through the *drm_irq_install()* and *drm_irq_uninstall()* functions. This only supports devices with a single interrupt on the main device stored in `drm_device.dev` and set as the device paramter in *drm_dev_alloc()*.

These IRQ helpers are strictly optional. Drivers which roll their own only need to set `drm_device.irq_enabled` to signal the DRM core that vblank interrupts are working. Since these helpers don't automatically clean up the requested interrupt like e.g. `devm_request_irq()` they're not really recommended.

int **drm_irq_install**(struct drm_device * *dev*, int *irq*)
install IRQ handler

Parameters

struct drm_device * dev DRM device

int irq IRQ number to install the handler for

Description

Initializes the IRQ related data. Installs the handler, calling the driver *drm_driver.irq_preinstall* and *drm_driver.irq_postinstall* functions before and after the installation.

This is the simplified helper interface provided for drivers with no special needs. Drivers which need to install interrupt handlers for multiple interrupts must instead set `drm_device.irq_enabled` to signal the DRM core that vblank interrupts are available.

irq must match the interrupt number that would be passed to `request_irq()`, if called directly instead of using this helper function.

drm_driver.irq_handler is called to handle the registered interrupt.

Return

Zero on success or a negative error code on failure.

int **drm_irq_uninstall**(struct drm_device * *dev*)
uninstall the IRQ handler

Parameters

struct drm_device * dev DRM device

Description

Calls the driver's *drm_driver.irq_uninstall* function and unregisters the IRQ handler. This should only be called by drivers which used *drm_irq_install()* to set up their interrupt handler. Other drivers must only reset `drm_device.irq_enabled` to false.

Note that for kernel modesetting drivers it is a bug if this function fails. The sanity checks are only to catch buggy user modesetting drivers which call the same function through an ioctl.

Return

Zero on success or a negative error code on failure.

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.

2.1.4 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.

drm_dma_handle_t * **drm_pci_alloc**(struct drm_device * *dev*, size_t *size*, size_t *align*)
Allocate a PCI consistent memory block, for DMA.

Parameters

struct drm_device * dev DRM device

size_t size size of block to allocate

size_t align alignment of block

Description

FIXME: This is a needless abstraction of the Linux dma-api and should be removed.

Return

A handle to the allocated memory block on success or NULL on failure.

void **drm_pci_free**(struct drm_device * *dev*, drm_dma_handle_t * *dmah*)
Free a PCI consistent memory block

Parameters

struct drm_device * dev DRM device

drm_dma_handle_t * dmah handle to memory block

Description

FIXME: This is a needless abstraction of the Linux dma-api and should be removed.

int **drm_get_pci_dev**(struct pci_dev * *pdev*, const struct pci_device_id * *ent*, struct drm_driver * *driver*)
Register a PCI device with the DRM subsystem

Parameters

struct pci_dev * pdev PCI device

const struct pci_device_id * ent entry from the PCI ID table that matches **pdev**

struct drm_driver * driver DRM device driver

Description

Attempt to gets inter module "drm" information. If we are first then register the character device and inter module information. Try and register, if we fail to register, backout previous work.

NOTE

This function is deprecated, please use `drm_dev_alloc()` and `drm_dev_register()` instead and remove your `drm_driver.load` callback.

Return

0 on success or a negative error code on failure.

int **drm_pci_init**(struct drm_driver * *driver*, struct pci_driver * *pdriver*)
Register matching PCI devices with the DRM subsystem

Parameters

struct drm_driver * driver DRM device driver

struct pci_driver * pdriver PCI device driver

Description

Initializes a drm_device structures, registering the stubs and initializing the AGP device.

NOTE

This function is deprecated. Modern modesetting drm drivers should use `pci_register_driver()` directly, this function only provides shadow-binding support for old legacy drivers on top of that core pci function.

Return

0 on success or a negative error code on failure.

void **drm_pci_exit**(struct *drm_driver* * *driver*, struct pci_driver * *pdriver*)
Unregister matching PCI devices from the DRM subsystem

Parameters

struct drm_driver * driver DRM device driver

struct pci_driver * pdriver PCI device driver

Description

Unregisters one or more devices matched by a PCI driver from the DRM subsystem.

NOTE

This function is deprecated. Modern modesetting drm drivers should use `pci_unregister_driver()` directly, this function only provides shadow-binding support for old legacy drivers on top of that core pci function.

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()*, and for drivers which use the CMA GEM helpers it's *drm_gem_cma_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 CMA based drivers there is the *DEFINE_DRM_GEM_CMA_FOPS()* macro to make this simpler.

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 nonblocking 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 {
  unsigned authenticated:1;
  unsigned stereo_allowed:1;
  unsigned universal_planes:1;
  unsigned atomic:1;
  unsigned is_master:1;
  struct drm_master * master;
  struct pid * pid;
  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.

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* .

master Master this node is currently associated with. 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.

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

pid Process that opened this file.

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_control_client**(const struct *drm_file* * *file_priv*)
is this an open file of the control node

Parameters

const struct drm_file * file_priv DRM file

Description

Control nodes are deprecated and in the process of getting removed from the DRM userspace API. Do not ever use!

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.

Return

0 on success or negative errno value on falure.

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.

Return

Always succeeds and returns 0.

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 iff 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. Therefore drivers also must set the `file_operation.llseek` to `no_llseek()`. 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.

Return

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

unsigned int **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 iff 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()*.

Return

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

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 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.

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

Return

0 on success or a negative error code on failure.

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.

Return

0 on success or a negative error code on failure.

void **drm_event_cancel_free**(struct drm_device * *dev*, struct *drm_pending_event* * *p*)
free a DRM event and release it's 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_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.

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(:c:type:`p`);
}
#endif

void some_other_function(...)
{
        struct drm_printer p = drm_info_printer(drm->dev);
        log_some_info(:c:type:`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.

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

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

2.4 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.

2.4.1 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.

2.4.2 Legacy DMA Services

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

CHAPTER

THREE

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 Maps (TTM) and Graphics Execution Manager (GEM). TTM was the first DRM memory manager to be developed and tried to be a one-size-fits-them all solution. It provides a single userspace API to 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 design background and information belongs here.

3.1.1 TTM initialization

Warning This section is outdated.

Drivers wishing to support TTM must pass a filled `ttm_bo_driver` structure to ttm_bo_device_init, together with an initialized global reference to the memory manager. The ttm_bo_driver structure contains several fields with function pointers for initializing the TTM, allocating and freeing memory, waiting for command completion and fence synchronization, and memory migration.

The `struct drm_global_reference` is made up of several fields:

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

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

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

See the radeon_ttm.c file for an example of usage.

int **drm_global_item_ref**(struct drm_global_reference * *ref*)
Initialize and acquire reference to memory object

Parameters

struct drm_global_reference * ref Object for initialization

Description

This initializes a memory object, allocating memory and calling the .:c:func:*init()* hook. Further calls will increase the reference count for that item.

Return

Zero on success, non-zero otherwise.

void **drm_global_item_unref**(struct drm_global_reference * *ref*)
Drop reference to memory object

Parameters

struct drm_global_reference * ref Object being removed

Description

Drop a reference to the memory object and eventually call the `release()` hook. The allocated object should be dropped in the `release()` hook or before calling this function

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. The caller must hold the `struct drm_device` struct_mutex lock when calling *drm_gem_object_get()*. As a convenience, GEM provides *drm_gem_object_put_unlocked()* functions that can be called without holding the lock.

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

void (*gem_free_object) (struct drm_gem_object *obj); Drivers are responsible for freeing all GEM object resources. 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 *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);
        int (*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_cma_get_unmapped_area()*. The mmap() routines will call this to get a proposed address for the mapping.

To use *drm_gem_cma_get_unmapped_area()*, drivers must fill the struct `struct file_operations` get_unmapped_area field with a pointer on *drm_gem_cma_get_unmapped_area()*.

More detailed information about get_unmapped_area can be found in Documentation/nommu-mmap.txt

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

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;
  uint32_t read_domains;
  uint32_t write_domain;
  uint32_t pending_read_domains;
  uint32_t pending_write_domain;
  struct dma_buf * dma_buf;
  struct dma_buf_attachment * import_attach;
};
```

Members

refcount Reference count of this object

Please use *drm_gem_object_get()* to acquire and *drm_gem_object_put()* or *drm_gem_object_put_unlocked()* 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 CMA 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.

read_domains Read memory domains. These monitor which caches contain read/write data related to the object. When transitioning from one set of domains to another, the driver is called to ensure that caches are suitably flushed and invalidated.

write_domain Corresponding unique write memory domain.

pending_read_domains While validating an exec operation, the new read/write domain values are computed here. They will be transferred to the above values at the point that any cache flushing occurs

pending_write_domain Write domain similar to **pending_read_domains**.

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_driver.gem_free_object* 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.

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.

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*)
raw function to release a GEM buffer object reference

Parameters

struct drm_gem_object * obj GEM buffer object

Description

This function is meant to be used by drivers which are not encumbered with `drm_device.struct_mutex` legacy locking and which are using the gem_free_object_unlocked callback. It avoids all the locking checks and locking overhead of *drm_gem_object_put()* and *drm_gem_object_put_unlocked()*.

Drivers should never call this directly in their code. Instead they should wrap it up into a `driver_gem_object_put(struct driver_gem_object *obj)` wrapper function, and use that. Shared code should never call this, to avoid breaking drivers by accident which still depend upon `drm_device.struct_mutex` locking.

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

Parameters

struct drm_gem_object * obj GEM buffer object

Description

This is a compatibility alias for *drm_gem_object_get()* and should not be used by new code.

void **__drm_gem_object_unreference**(struct *drm_gem_object* * *obj*)
raw function to release a GEM buffer object reference

Parameters

struct drm_gem_object * obj GEM buffer object

Description

This is a compatibility alias for *__drm_gem_object_put()* and should not be used by new code.

void **drm_gem_object_unreference_unlocked**(struct *drm_gem_object* * *obj*)
release a GEM buffer object reference

Parameters

struct drm_gem_object * obj GEM buffer object

Description

This is a compatibility alias for *drm_gem_object_put_unlocked()* and should not be used by new code.

void **drm_gem_object_unreference**(struct *drm_gem_object* * *obj*)
release a GEM buffer object reference

Parameters

struct drm_gem_object * obj GEM buffer object

Description

This is a compatibility alias for *drm_gem_object_put()* and should not be used by new code.

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.

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_destroy**(struct *drm_file* * *file*, struct drm_device * *dev*, uint32_t *handle*)
dumb fb callback helper for gem based drivers

Parameters

struct drm_file * file drm file-private structure to remove the dumb handle from

struct drm_device * dev corresponding drm_device

uint32_t handle the dumb handle to remove

Description

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

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 pionter 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.

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_init_object()` via `mapping_set_gfp_mask()`. shmem-core takes care to keep pages in the required zone during swap-in.

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

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

Parameters

struct drm_file * filp DRM file private date

u32 handle userspace handle

Return

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

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 invers 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. Must be called holding `drm_device.struct_mutex`.

Frees the object

void **drm_gem_object_put_unlocked**(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**. Callers must not hold the `drm_device.struct_mutex` lock when calling this function.

See also *__drm_gem_object_put()*.

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

Parameters

struct drm_gem_object * obj GEM buffer object

Description

This releases a reference to **obj**. Callers must hold the `drm_device.struct_mutex` lock when calling this function, even when the driver doesn't use `drm_device.struct_mutex` for anything.

For drivers not encumbered with legacy locking use *drm_gem_object_put_unlocked()* instead.

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_vm_ops provided by the driver. Depending on their requirements, drivers can either provide a fault handler in their gem_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 gem_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.

3.2.9 GEM CMA Helper Functions Reference

The Contiguous Memory Allocator reserves a pool of memory at early boot that is used to service requests for large blocks of contiguous memory.

The DRM GEM/CMA helpers use this allocator as a means to provide buffer objects that are physically contiguous in memory. This is useful for display drivers that are unable to map scattered buffers via an IOMMU.

struct **drm_gem_cma_object**
GEM object backed by CMA memory allocations

Definition

```
struct drm_gem_cma_object {
  struct drm_gem_object base;
  dma_addr_t paddr;
  struct sg_table * sgt;
```

```
  void * vaddr;
};
```

Members

base base GEM object

paddr physical address of the backing memory

sgt scatter/gather table for imported PRIME buffers

vaddr kernel virtual address of the backing memory

DEFINE_DRM_GEM_CMA_FOPS(*name*)
macro to generate file operations for CMA drivers

Parameters

name name for the generated structure

Description

This macro autogenerates a suitable `struct file_operations` for CMA 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_cma_object* * **drm_gem_cma_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 CMA GEM object and allocates a contiguous chunk of memory as backing store. The backing memory has the writecombine attribute set.

Return

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

void **drm_gem_cma_free_object**(struct *drm_gem_object* * *gem_obj*)
free resources associated with a CMA GEM object

Parameters

struct drm_gem_object * gem_obj GEM object to free

Description

This function frees the backing memory of the CMA GEM object, cleans up the GEM object state and frees the memory used to store the object itself. Drivers using the CMA helpers should set this as their *drm_driver.gem_free_object* callback.

int **drm_gem_cma_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_cma_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_cma_dumb_create_internal()` function.

Return

0 on success or a negative error code on failure.

int **drm_gem_cma_dumb_map_offset**(struct *drm_file* * *file_priv*, struct drm_device * *drm*, u32 *handle*, u64 * *offset*)
return the fake mmap offset for a CMA GEM object

Parameters

struct drm_file * file_priv DRM file-private structure containing the GEM object

struct drm_device * drm DRM device

u32 handle GEM object handle

u64 * offset return location for the fake mmap offset

Description

This function look up an object by its handle and returns the fake mmap offset associated with it. Drivers using the CMA helpers should set this as their `drm_driver.dumb_map_offset` callback.

Return

0 on success or a negative error code on failure.

int **drm_gem_cma_mmap**(struct file * *filp*, struct vm_area_struct * *vma*)
memory-map a CMA GEM object

Parameters

struct file * filp file 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 CMA objects: In addition to the usual GEM VMA setup it immediately faults in the entire object instead of using on-demaind faulting. Drivers which employ the CMA helpers should use this function as their ->:c:func:*mmap()* handler in the DRM device file's file_operations structure.

Instead of directly referencing this function, drivers should use the *DEFINE_DRM_GEM_CMA_FOPS()*.macro.

Return

0 on success or a negative error code on failure.

unsigned long **drm_gem_cma_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_cma_describe**(struct *drm_gem_cma_object* * *cma_obj*, struct seq_file * *m*)
describe a CMA GEM object for debugfs

Parameters

struct drm_gem_cma_object * cma_obj CMA GEM object

struct seq_file * m debugfs file handle

Description

This function can be used to dump a human-readable representation of the CMA GEM object into a synthetic file.

struct sg_table * **drm_gem_cma_prime_get_sg_table**(struct *drm_gem_object* * *obj*)
provide a scatter/gather table of pinned pages for a CMA GEM object

Parameters

struct drm_gem_object * obj GEM object

Description

This function exports a scatter/gather table suitable for PRIME usage by calling the standard DMA mapping API. Drivers using the CMA helpers should set this as their *drm_driver.gem_prime_get_sg_table* callback.

Return

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

struct *drm_gem_object* * **drm_gem_cma_prime_import_sg_table**(struct drm_device * *dev*, struct dma_buf_attachment * *attach*, struct sg_table * *sgt*)
produce a CMA 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 CMA 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_cma_prime_mmap**(struct *drm_gem_object* * *obj*, struct vm_area_struct * *vma*)
memory-map an exported CMA GEM object

Parameters

struct drm_gem_object * obj GEM object

struct vm_area_struct * vma VMA for the area to be mapped

Description

This function maps a buffer imported via DRM PRIME into a userspace process's address space. Drivers that use the CMA helpers should set this as their *drm_driver.gem_prime_mmap* callback.

Return

0 on success or a negative error code on failure.

void * **drm_gem_cma_prime_vmap**(struct *drm_gem_object* * *obj*)
map a CMA GEM object into the kernel's virtual address space

Parameters

struct drm_gem_object * obj GEM object

Description

This function maps a buffer exported via DRM PRIME into the kernel's virtual address space. Since the CMA buffers are already mapped into the kernel virtual address space this simply returns the cached virtual address. Drivers using the CMA helpers should set this as their DRM driver's *drm_driver.gem_prime_vmap* callback.

Return

The kernel virtual address of the CMA GEM object's backing store.

void **drm_gem_cma_prime_vunmap**(struct *drm_gem_object* * *obj*, void * *vaddr*)
unmap a CMA GEM object from the kernel's virtual address space

Parameters

struct drm_gem_object * obj GEM object

void * vaddr kernel virtual address where the CMA GEM object was mapped

Description

This function removes a buffer exported via DRM PRIME from the kernel's virtual address space. This is a no-op because CMA buffers cannot be unmapped from kernel space. Drivers using the CMA helpers should set this as their *drm_driver.gem_prime_vunmap* callback.

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**(struct drm_vma_offset_node * *node*)
Return start address for page-based addressing

Parameters

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 mmaps

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()` callbacks.

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()`.

Example

```
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);
```

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)

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 iff **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 Driver Interface

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 must set the DRIVER_PRIME bit in the struct *struct drm_driver* driver_features field, and implement the prime_handle_to_fd and prime_fd_to_handle operations.

int (*prime_handle_to_fd)(struct drm_device *dev, struct drm_file *file_priv, uint32_t handle, uint32_t flags, int *prime_fd); int (*prime_fd_to_handle)(struct drm_device *dev, struct drm_file *file_priv, int prime_fd, uint32_t *handle); Those two operations convert a handle to a PRIME file descriptor and vice versa. Drivers must use the kernel dma-buf buffer sharing framework to manage the PRIME file descriptors. Similar to the mode setting API PRIME is agnostic to the underlying buffer object manager, as long as handles are 32bit unsigned integers.

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

struct dma_buf * (*gem_prime_export)(struct drm_device *dev, struct drm_gem_object *obj, int flags); struct drm_gem_object * (*gem_prime_import)(struct drm_device *dev, struct dma_buf *dma_buf); These two operations are mandatory for GEM drivers that support PRIME.

3.4.2 PRIME Helper Functions

Drivers can implement **gem_prime_export** and **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 six lower-level driver callbacks:

Export callbacks:

- **gem_prime_pin** (optional): prepare a GEM object for exporting
- **gem_prime_get_sg_table**: provide a scatter/gather table of pinned pages
- **gem_prime_vmap**: vmap a buffer exported by your driver
- **gem_prime_vunmap**: vunmap a buffer exported by your driver
- **gem_prime_mmap** (optional): mmap a buffer exported by your driver

Import callback:

- **gem_prime_import_sg_table** (import): produce a GEM object from another driver's scatter/gather table

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()*.

struct dma_buf * **drm_gem_prime_export**(struct drm_device * *dev*, struct *drm_gem_object* * *obj*, int *flags*)

helper library implementation of the export callback

Parameters

struct drm_device * dev drm_device to export from

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 gem_prime_export functions for GEM drivers using the PRIME helpers.

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 gem_prime_export driver callback.

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 dev->dev for attaching via dma_buf.

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.

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 dev to export the buffer from

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 gem_import_export driver callback.

struct sg_table * **drm_prime_pages_to_sg**(struct page ** *pages*, unsigned int *nr_pages*)
converts a page array into an sg list

Parameters

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.

int **drm_prime_sg_to_page_addr_arrays**(struct sg_table * *sgt*, struct page ** *pages*, dma_addr_t * *addrs*, int *max_pages*)
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 page array in

dma_addr_t * addrs optional array to store the dma bus address of each page

int max_pages size of both the passed-in arrays

Description

Exports an sg table into an array of pages and addresses. This is currently required by the TTM driver in order to do correct fault handling.

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** 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(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 on 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 candiates 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(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.

Description

The *struct drm_mm* range manager supports finding a suitable modes using a number of search trees. These trees are oranised 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 funcions. 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(*mm*)
list of nodes under the drm_mm range manager

Parameters

mm the struct drm_mm range manger

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(*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(*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(*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_generic()` 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(*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 its 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 size end 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 Cache Handling

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 scather-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.

3.7 DRM Sync Objects

DRM synchronisation objects (syncobj) are a persistent objects, that contain an optional fence. The fence can be updated with a new fence, or be NULL.

syncobj's can be export to fd's and back, these fd's are opaque and have no other use case, except passing the syncobj between processes.

Their primary use-case is to implement Vulkan fences and semaphores.

syncobj have a kref reference count, but also have an optional file. The file is only created once the syncobj is exported. The file takes a reference on the kref.

struct **drm_syncobj**
sync object.

Definition

```
struct drm_syncobj {
  struct kref refcount;
  struct dma_fence * fence;
  struct file * file;
};
```

Members

refcount Reference count of this object.

fence NULL or a pointer to the fence bound to this object.

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 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 *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.

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.

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.

CHAPTER

FOUR

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. 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 unecessarily 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. Futhermore 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 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*.

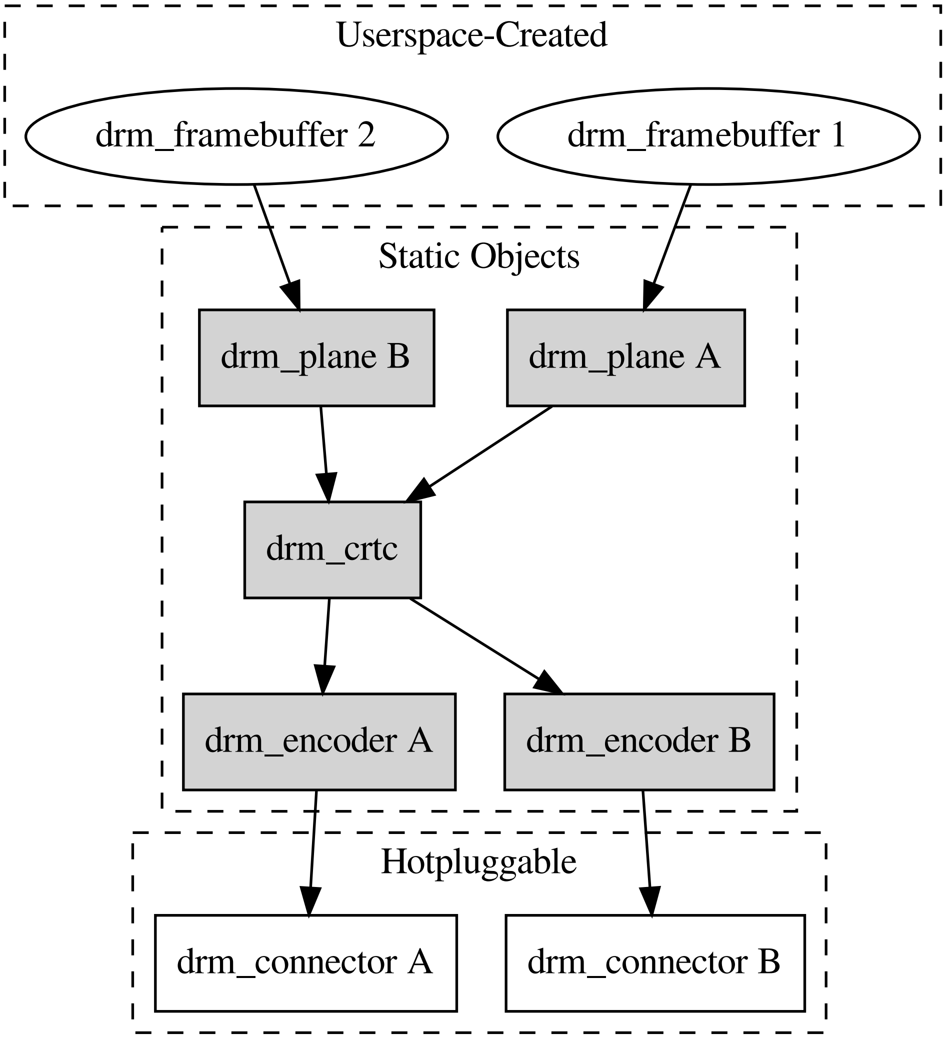

Fig. 4.1: KMS Display Pipeline Overview

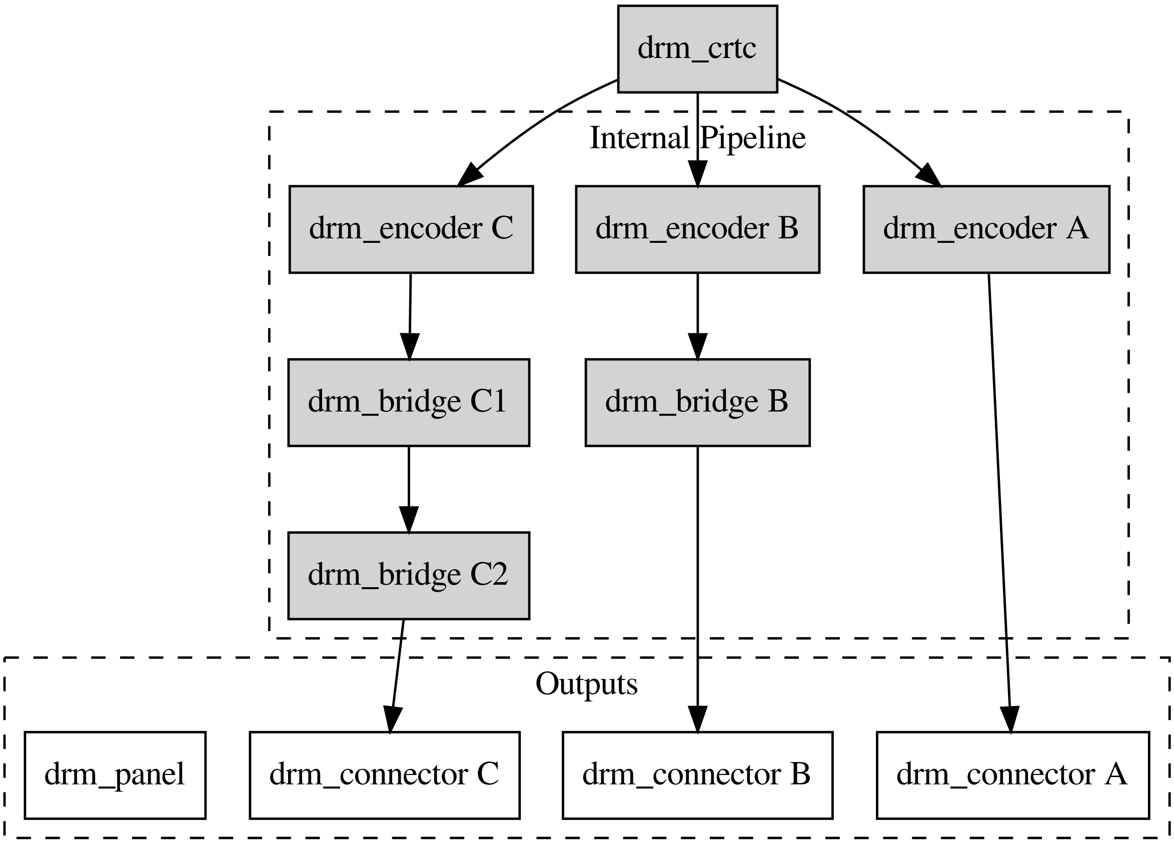

Fig. 4.2: KMS Output Pipeline

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);
  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.

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 with the helpers can call drm_fb_helper_hotplug_changed from this hook to inform the fbdev helper of output changes.

FIXME:

Except that there's no vtable for device-level helper callbacks there's no reason this is a core function.

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 updated is requested and there is an earlier updated 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**.

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.

`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.

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 crtc_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;
  int num_encoder;
  struct list_head encoder_list;
  int num_overlay_plane;
  int num_total_plane;
  struct list_head plane_list;
  int num_crtc;
  struct list_head crtc_list;
  struct list_head property_list;
  int min_width;
  int min_height;
  int max_width;
  int max_height;
  const struct drm_mode_config_funcs * funcs;
  resource_size_t fb_base;
  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_active;
  struct drm_property * prop_mode_id;
  struct drm_property * dvi_i_subconnector_property;
  struct drm_property * dvi_i_select_subconnector_property;
  struct drm_property * tv_subconnector_property;
  struct drm_property * tv_select_subconnector_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 * 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;
  uint32_t preferred_depth;
  uint32_t prefer_shadow;
  bool async_page_flip;
  bool allow_fb_modifiers;
  uint32_t cursor_width;
  uint32_t cursor_height;
  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 it's 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 **crtc_idr** and **tile_idr**.

`crtc_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**.

`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.

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_overlay_plane Number of overlay planes on this device, excluding primary and cursor planes.

Track number of overlay planes separately from number of total planes. By default we only advertise overlay planes to userspace; if userspace sets the "universal plane" capability bit, we'll go ahead and expose all planes. 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/curso) 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 CRTCs 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.

min_width minimum pixel width on this device

min_height minimum pixel height on this device

max_width maximum pixel width on this device

max_height maximum pixel height on this device

funcs core driver provided mode setting functions

fb_base base address of the framebuffer

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 is being shown on.

prop_crtc_y Default atomic plane property for the plane destination position in the *drm_crtc* is is being shown on.

prop_crtc_w Default atomic plane property for the plane destination position in the *drm_crtc* is is being shown on.

prop_crtc_h Default atomic plane property for the plane destination position in the *drm_crtc* is 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_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.

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.

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.

tv_mode_property Optional TV property to select the output TV mode.

tv_left_margin_property Optional TV property to set the left margin.

tv_right_margin_property Optional TV property to set the right margin.

tv_top_margin_property Optional TV property to set the right margin.

tv_bottom_margin_property Optional TV property to set the right margin.

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.

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.

preferred_depth preferred RBG pixel depth, used by fb helpers

prefer_shadow hint to userspace to prefer shadow-fb rendering

async_page_flip Does this device support async flips on the primary plane?

allow_fb_modifiers Whether the driver supports fb modifiers in the ADDFB2.1 ioctl call.

cursor_width hint to userspace for max cursor width

cursor_height hint to userspace for max cursor height

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.

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.

void **drm_mode_config_init**(struct drm_device * *dev*)
initialize DRM mode_configuration structure

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.

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 CRTCs associated with this DRM device, then free up the framebuffers 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: cleanup any dangling user buffer objects too

4.3 Modeset Base Object Abstraction

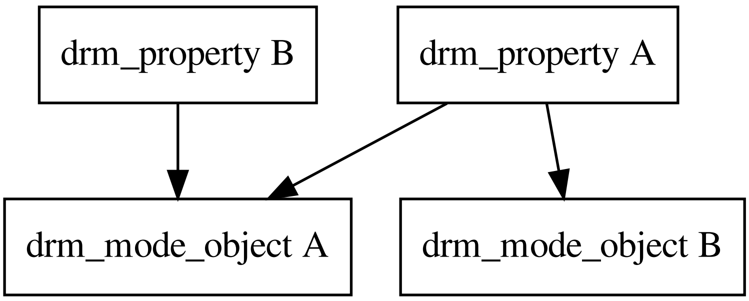

Fig. 4.3: Mode Objects and Properties

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;
  uint64_t values;
};
```

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.

void **drm_mode_object_reference**(struct *drm_mode_object* * *obj*)
acquire a mode object reference

Parameters

struct drm_mode_object * obj DRM mode object

Description

This is a compatibility alias for *drm_mode_object_get()* and should not be used by new code.

void **drm_mode_object_unreference**(struct *drm_mode_object* * *obj*)
release a mode object reference

Parameters

struct drm_mode_object * obj DRM mode object

Description

This is a compatibility alias for *drm_mode_object_put()* and should not be used by new code.

struct *drm_mode_object* * **drm_mode_object_find**(struct drm_device * *dev*, uint32_t *id*, uint32_t *type*)
look up a drm object with static lifetime

Parameters

struct drm_device * dev drm device

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 callind *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.

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 softare 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.

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.
- 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 CRTCs should not interfere, and not get stalled due to output routing changing on different CRTCs.

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 CRTCs 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 embeddeding, or add entirely new state structures for their globally shared hardware functions.

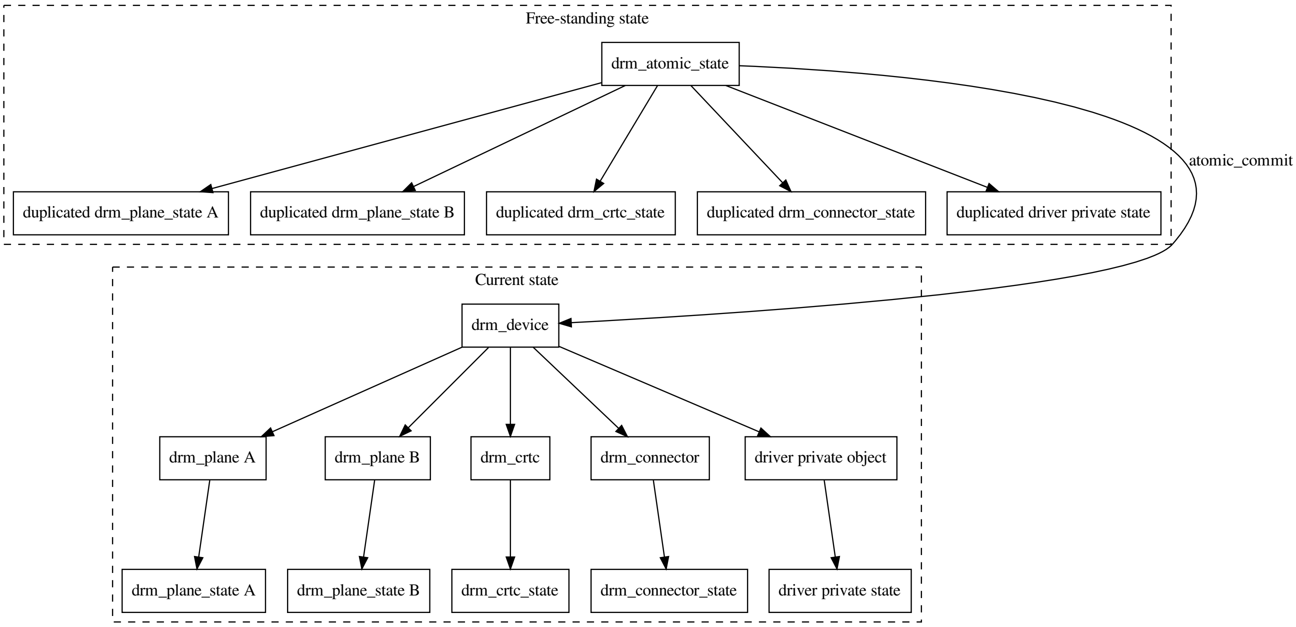

Fig. 4.4: Mode Objects and Properties

- An atomic update is assembled and validated as an entirely free-standing pile of structures within the *drm_atomic_state* container. Again drivers can subclass that container for their own state structure tracking needs. 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.

Read on in this chapter, and also in *Atomic Modeset Helper Functions Reference* for more detailed coverage of specific topics.

4.4.1 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;
};
```

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 signaled 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 singalled. Note that for most hardware, in most cases this happens after **hw_done** is signalled.

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 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.

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.

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.

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.

struct **drm_private_state_funcs**
atomic state functions for private objects

Definition

```
struct drm_private_state_funcs {
  void *(* duplicate_state) (struct drm_atomic_state *state, void *obj);
  void (* swap_state) (void *obj, void **obj_state_ptr);
  void (* destroy_state) (void *obj_state);
};
```

Members

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.

swap_state This function swaps the existing state of a private object **obj** with it's newly created state, the pointer to which is passed as **obj_state_ptr**.

destroy_state Frees the private object state created with **duplicate_state**.

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 it's drm_private_state_funcs struct to *drm_atomic_get_private_obj_state()*.

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;
  struct __drm_planes_state * planes;
  struct __drm_crtcs_state * crtcs;
  int num_connector;
  struct __drm_connnectors_state * connectors;
  int num_private_objs;
  struct __drm_private_objs_state * private_objs;
  struct drm_modeset_acquire_ctx * acquire_ctx;
  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

legacy_cursor_update hint to enforce legacy cursor IOCTL semantics

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

commit_work Work item which can be used by the driver or helpers to execute the commit without blocking.

void **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**.

void **drm_crtc_commit_put**(struct *drm_crtc_commit* * *commit*)
release a reference to the CRTC commmit

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**(struct *drm_atomic_state* * *state*, struct *drm_crtc* * *crtc*)
get crtc state, if it exists

Parameters

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**(struct *drm_atomic_state* * *state*, struct *drm_crtc* * *crtc*)
get old crtc state, if it exists

Parameters

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**(struct *drm_atomic_state* * *state*, struct *drm_crtc* * *crtc*)

get new crtc state, if it exists

Parameters

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**(struct *drm_atomic_state* * *state*, struct *drm_plane* * *plane*)

get plane state, if it exists

Parameters

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**(struct *drm_atomic_state* * *state*, struct *drm_plane* * *plane*)

get plane state, if it exists

Parameters

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**(struct *drm_atomic_state* * *state*, struct *drm_plane* * *plane*)

get plane state, if it exists

Parameters

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**(struct *drm_atomic_state* * *state*, struct *drm_connector* * *connector*)

get connector state, if it exists

Parameters

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**(struct *drm_atomic_state* * *state*, struct *drm_connector* * *connector*)

get connector state, if it exists

Parameters

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**(struct *drm_atomic_state* * *state*, struct *drm_connector* * *connector*)

get connector state, if it exists

Parameters

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**(struct *drm_atomic_state* * *state*, struct *drm_plane* * *plane*)

get current plane state

Parameters

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.

Return

Read-only pointer to the current plane state.

for_each_connector_in_state(*__state*, *connector*, *connector_state*, *__i*)
iterate over all connectors in an atomic update

Parameters

__state *struct drm_atomic_state* pointer

connector *struct drm_connector* iteration cursor

connector_state *struct drm_connector_state* iteration cursor

__i int iteration cursor, for macro-internal use

Description

This iterates over all connectors in an atomic update. Note that before the software state is committed (by calling *drm_atomic_helper_swap_state()*, this points to the new state, while afterwards it points to the old state. Due to this tricky confusion this macro is deprecated.

FIXME:

Replace all usage of this with one of the explicit iterators below and then remove this macro.

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(*__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(*__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_crtc_in_state(*__state*, *crtc*, *crtc_state*, *__i*)
iterate over all connectors in an atomic update

Parameters

__state *struct drm_atomic_state* pointer

crtc *struct drm_crtc* iteration cursor

crtc_state *struct drm_crtc_state* iteration cursor

__i int iteration cursor, for macro-internal use

Description

This iterates over all CRTCs in an atomic update. Note that before the software state is committed (by calling *drm_atomic_helper_swap_state()*, this points to the new state, while afterwards it points to the old state. Due to this tricky confusion this macro is deprecated.

FIXME:

Replace all usage of this with one of the explicit iterators below and then remove this macro.

for_each_oldnew_crtc_in_state(*__state*, *crtc*, *old_crtc_state*, *new_crtc_state*, *__i*)
iterate over all CRTCs 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 CRTCs 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(*__state*, *crtc*, *old_crtc_state*, *__i*)
iterate over all CRTCs 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 CRTCs 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(*__state*, *crtc*, *new_crtc_state*, *__i*)
iterate over all CRTCs 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 CRTCs 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_plane_in_state(*__state*, *plane*, *plane_state*, *__i*)
iterate over all planes in an atomic update

Parameters

__state *struct drm_atomic_state* pointer

plane *struct drm_plane* iteration cursor

plane_state *struct drm_plane_state* iteration cursor

__i int iteration cursor, for macro-internal use

Description

This iterates over all planes in an atomic update. Note that before the software state is committed (by calling *drm_atomic_helper_swap_state()*, this points to the new state, while afterwards it points to the old state. Due to this tricky confusion this macro is deprecated.

FIXME:

Replace all usage of this with one of the explicit iterators below and then remove this macro.

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_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(*__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_private_obj(*__state*, *obj*, *obj_state*, *__i*, *__funcs*)
iterate over all private objects

Parameters

__state *struct drm_atomic_state* pointer

obj private object iteration cursor

obj_state private object state iteration cursor

__i int iteration cursor, for macro-internal use

__funcs *struct drm_private_state_funcs* iteration cursor

Description

This macro iterates over the array containing private object data in atomic state

for_each_private_obj(*__state*, *obj_funcs*, *obj*, *obj_state*, *__i*, *__funcs*)
iterate over a specify type of private object

Parameters

__state *struct drm_atomic_state* pointer

obj_funcs *struct drm_private_state_funcs* function table to filter private objects

obj private object iteration cursor

obj_state private object state iteration cursor

__i int iteration cursor, for macro-internal use

__funcs *struct drm_private_state_funcs* iteration cursor

Description

This macro iterates over the private objects state array while filtering the objects based on the vfunc table that is passed as **obj_funcs**. New macros can be created by passing in the vfunc table associated with a specific private object.

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: planes_changed, mode_changed and active_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.

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 is useful for drivers that subclass the atomic state.

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 is useful for drivers that subclass the atomic state.

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 is useful for drivers that subclass the atomic state.

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, crtcs 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.

Return

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.

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_crtc_set_property**(struct *drm_crtc* * *crtc*, struct *drm_crtc_state* * *state*, struct *drm_property* * *property*, uint64_t *val*)

set property on CRTC

Parameters

struct drm_crtc * crtc the drm CRTC to set a property on

struct drm_crtc_state * state the state object to update with the new property value

struct drm_property * property the property to set

uint64_t val the new property value

Description

This function handles generic/core properties and calls out to driver's *drm_crtc_funcs.atomic_set_property* for driver properties. To ensure consistent behavior you must call this function rather than the driver hook directly.

Return

Zero on success, error code on failure

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.

Return

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.

int **drm_atomic_plane_set_property**(struct *drm_plane* * *plane*, struct *drm_plane_state* * *state*, struct *drm_property* * *property*, uint64_t *val*)

set property on plane

Parameters

struct drm_plane * plane the drm plane to set a property on

struct drm_plane_state * state the state object to update with the new property value

struct drm_property * property the property to set

uint64_t val the new property value

Description

This function handles generic/core properties and calls out to driver's *drm_plane_funcs.atomic_set_property* for driver properties. To ensure consistent behavior you must call this function rather than the driver hook directly.

Return

Zero on success, error code on failure

void * **drm_atomic_get_private_obj_state**(struct *drm_atomic_state* * *state*, void * *obj*, const struct *drm_private_state_funcs* * *funcs*)

get private object state

Parameters

struct drm_atomic_state * state global atomic state

void * obj private object to get the state for

const struct drm_private_state_funcs * funcs pointer to the struct of function pointers that identify the object type

Description

This function returns the private object state for the given private object, allocating the state if needed. It does not grab any locks as the caller is expected to care of any required locking.

Return

Either the allocated state or the error code encoded into a pointer.

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.

Return

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.

int **drm_atomic_connector_set_property**(struct *drm_connector* * *connector*, struct *drm_connector_state* * *state*, struct *drm_property* * *property*, uint64_t *val*)

set property on connector.

Parameters

struct drm_connector * connector the drm connector to set a property on

struct drm_connector_state * state the state object to update with the new property value

struct drm_property * property the property to set

uint64_t val the new property value

Description

This function handles generic/core properties and calls out to driver's *drm_connector_funcs.atomic_set_property* for driver properties. To ensure consistent behavior you must call this function rather than the driver hook directly.

Return

Zero on success, error code on failure

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.

void **drm_atomic_set_fence_for_plane**(struct *drm_plane_state* * *plane_state*, struct dma_fence * *fence*)
set fence for plane

Parameters

struct drm_plane_state * plane_state atomic state object for the plane

struct dma_fence * fence dma_fence to use for the plane

Description

Helper to setup the plane_state fence in case it is not set yet. By using this drivers doesn't need to worry if the user choose implicit or explicit fencing.

This function will not set the fence to the state if it was set via explicit fencing interfaces on the atomic ioctl. In that case it will drop the reference to the fence as we are not storing it anywhere. Otherwise, if `drm_plane_state.fence` is not set this function we just set it with the received implicit fence. In both cases this function consumes a reference for **fence**.

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.

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 seralization 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 parallism 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.

void **drm_atomic_legacy_backoff**(struct *drm_atomic_state* * *state*)
locking backoff for legacy ioctls

Parameters

struct drm_atomic_state * state atomic state

Description

This function should be used by legacy entry points which don't understand -EDEADLK semantics. For simplicity this one will grab all modeset locks after the slowpath completed.

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_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 *drm_modeset_lock_all()*, or if this is called from error irq handler, it should not be enabled by default. (Ie. if you are debugging errors you might not care that this is racey. But calling this without all modeset locks held is not inherently safe.)

void **drm_atomic_clean_old_fb**(struct drm_device * *dev*, unsigned *plane_mask*, int *ret*)

- Unset old_fb pointers and set plane->fb pointers.

Parameters

struct drm_device * dev drm device to check.

unsigned plane_mask plane mask for planes that were updated.

int ret return value, can be -EDEADLK for a retry.

Description

Before doing an update *drm_plane.old_fb* is set to *drm_plane.fb*, but before dropping the locks old_fb needs to be set to NULL and plane->fb updated. This is a common operation for each atomic update, so this call is split off as a helper.

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* 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;
  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;
  u32 pageflip_flags;
  struct drm_pending_vblank_event * event;
  struct drm_atomic_state * state;
};
```

Members

crtc backpointer to the CRTC

enable whether the CRTC should be enabled, gates all other state

active whether the CRTC is actively displaying (used for DPMS)

planes_changed planes on this crtc are updated

mode_changed **mode** or **enable** has been changed

active_changed **active** has been toggled.

connectors_changed connectors to this crtc have been updated

zpos_changed zpos values of planes on this crtc have been updated

color_mgmt_changed color management properties have changed (degamma or gamma LUT or CSC matrix)

plane_mask bitmask of (1 << drm_plane_index(plane)) of attached planes

connector_mask bitmask of (1 << drm_connector_index(connector)) of attached connectors

encoder_mask bitmask of (1 << drm_encoder_index(encoder)) of attached encoders

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. It is purely driver implementation defined what exactly this adjusted mode means. Usually it is 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**

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`.

target_vblank Target vertical blank period when a page flip should take effect.

pageflip_flags DRM_MODE_PAGE_FLIP_* flags, as passed to the page flip ioctl. Zero in any other case.

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 send 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()* somewhen after *drm_crtc_vblank_off()* has been called.
- For a CRTC which is enabled at the end of the commit (even when it undergoes an 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.

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.

state backpointer to global drm_atomic_state

Description

Note that the distinction between **enable** and **active** is rather subtile: 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 CRTCs 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, size_t *values_cnt);
  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);
};
```

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.

NOTE:

Drivers that support gamma tables and also fbdev emulation through the provided helper library need to take care to fill out the gamma hooks for both. Currently there's a bit an unfortunate duplication going on, which should eventually be unified to just one set of hooks.

destroy Clean up plane 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.

Drivers implementing atomic modeset should use *drm_atomic_helper_crtc_set_property()* to implement this hook.

This callback is optional if the driver does not support any legacy driver-private properties.

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.

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

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 torndown.

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.

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.

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.

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.

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_crtc_state * state;
  struct list_head commit_list;
  spinlock_t commit_lock;
#ifdef CONFIG_DEBUG_FS
  struct dentry * debugfs_entry;
#endif
  struct drm_crtc_crc crc;
  unsigned int fence_context;
  spinlock_t fence_lock;
  unsigned long fence_seqno;
  char timeline_name;
};
```

Members

dev parent DRM device

port OF node used by `drm_of_find_possible_crtcs()`

head list management

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

cursor cursor plane for this CRTC

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

cursor_y current y position of the cursor, used for universal cursor planes

enabled is this CRTC enabled?

mode current mode timings

hwmode mode timings as programmed to hw regs

x x position on screen

y y position on screen

funcs CRTC control functions

gamma_size size of gamma ramp

gamma_store gamma ramp values

helper_private mid-layer private data

properties property tracking for this CRTC

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_crtc_in_state()*, *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 doesn't hold its own full reference, but burrows it from the ongoing commit. Commit entries must be removed from this list once the commit is fully completed, but before it's correspoding *drm_atomic_state* gets destroyed.

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.

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*.

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 an encoder's possible_crtcs field.

struct *drm_crtc* * **drm_crtc_find**(struct drm_device * *dev*, uint32_t *id*)
look up a CRTC object from its ID

Parameters

struct drm_device * dev DRM device

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(*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**.

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_force_disable**(struct *drm_crtc* * *crtc*)
Forcibly turn off a CRTC

Parameters

struct drm_crtc * crtc CRTC to turn off

Note

This should only be used by non-atomic legacy drivers.

Return

Zero on success, error code on failure.

int **drm_crtc_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.

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

Inits 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.

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

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 priviledged 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.

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;
  const struct drm_format_info * format;
  const struct drm_framebuffer_funcs * funcs;
  unsigned int pitches;
  unsigned int offsets;
  uint64_t modifier;
  unsigned int width;
  unsigned int height;
  int flags;
  int hot_x;
  int hot_y;
  struct list_head filp_head;
};
```

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.

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 laytou per **modifier**. It 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 it **offsets** must at least be tile-size aligned, but hardware often has stricter requirements.

This should not be used to specifiy 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*.

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.

void **drm_framebuffer_reference**(struct *drm_framebuffer* * *fb*)
acquire a framebuffer reference

Parameters

struct drm_framebuffer * fb DRM framebuffer

Description

This is a compatibility alias for *drm_framebuffer_get()* and should not be used by new code.

void **drm_framebuffer_unreference**(struct *drm_framebuffer* * *fb*)
release a framebuffer reference

Parameters

struct drm_framebuffer * fb DRM framebuffer

Description

This is a compatibility alias for *drm_framebuffer_put()* and should not be used by new code.

uint32_t **drm_framebuffer_read_refcount**(struct *drm_framebuffer* * *fb*)
read the framebuffer reference count.

Parameters

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.

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*, uint32_t *id*)
look up a drm framebuffer and grab a reference

Parameters

struct drm_device * dev drm device

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 it's 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 info 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 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 CRTCs 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

struct **drm_format_info**
information about a DRM format

Definition

```
struct drm_format_info {
  u32 format;
  u8 depth;
  u8 num_planes;
  u8 cpp;
  u8 hsub;
  u8 vsub;
};
```

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)

cpp Number of bytes per pixel (per plane)

hsub Horizontal chroma subsampling factor

vsub Vertical chroma subsampling factor

struct **drm_format_name_buf**
name of a DRM format

Definition

```
struct drm_format_name_buf {
  char str;
};
```

Members

str string buffer containing the format name

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.

const char * **drm_get_format_name**(uint32_t *format*, struct *drm_format_name_buf* * *buf*)
fill a string with a drm fourcc format's name

Parameters

uint32_t format format to compute name of

struct drm_format_name_buf * buf caller-supplied buffer

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.

int **drm_format_num_planes**(uint32_t *format*)
get the number of planes for format

Parameters

uint32_t format pixel format (DRM_FORMAT_*)

Return

The number of planes used by the specified pixel format.

int **drm_format_plane_cpp**(uint32_t *format*, int *plane*)
determine the bytes per pixel value

Parameters

uint32_t format pixel format (DRM_FORMAT_*)

int plane plane index

Return

The bytes per pixel value for the specified plane.

int **drm_format_horz_chroma_subsampling**(uint32_t *format*)
get the horizontal chroma subsampling factor

Parameters

uint32_t format pixel format (DRM_FORMAT_*)

Return

The horizontal chroma subsampling factor for the specified pixel format.

int **drm_format_vert_chroma_subsampling**(uint32_t *format*)
get the vertical chroma subsampling factor

Parameters

uint32_t format pixel format (DRM_FORMAT_*)

Return

The vertical chroma subsampling factor for the specified pixel format.

int **drm_format_plane_width**(int *width*, uint32_t *format*, int *plane*)
width of the plane given the first plane

Parameters

int width width of the first plane

uint32_t format pixel format

int plane plane index

Return

The width of **plane**, given that the width of the first plane is **width**.

int **drm_format_plane_height**(int *height*, uint32_t *format*, int *plane*)
height of the plane given the first plane

Parameters

int height height of the first plane

uint32_t format pixel format

int plane plane index

Return

The height of **plane**, given that the height of the first plane is **height**.

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*, *drm_driver.dumb_destroy* and *drm_driver.dumb_map_offset* operations. See there 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 overlayed 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 are 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*.

To create a plane, a KMS drivers allocates and zeroes an instances of *struct drm_plane* (possibly as part of a larger structure) and registers it with a call to *drm_universal_plane_init()*.

Cursor and overlay planes are optional. All drivers should provide one primary plane per CRTC to avoid surprising userspace too much. See enum drm_plane_type for a more in-depth discussion of these special uapi-relevant plane types. Special planes are associated with their CRTC by calling *drm_crtc_init_with_planes()*.

The type of a plane is exposed in the immutable "type" enumeration property, which has one of the following values: "Overlay", "Primary", "Cursor".

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;
  uint32_t crtc_h;
  uint32_t src_x;
  uint32_t src_y;
  uint32_t src_h;
  uint32_t src_w;
  unsigned int rotation;
  unsigned int zpos;
  unsigned int normalized_zpos;
  struct drm_rect src;
  struct drm_rect dst;
  bool visible;
  struct drm_atomic_state * state;
};
```

Members

plane backpointer to the plane

crtc Currently bound CRTC, NULL if disabled. Do not this write 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**. Do not write this directly, use *drm_atomic_set_fence_for_plane()*

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)

src_y upper position of visible portion of plane within plane (in 16.16)

src_h height of visible portion of plane (in 16.16)

src_w width of visible portion of plane (in 16.16)

rotation rotation of the plane

zpos priority of the given plane on crtc (optional) Note that multiple active planes on the same crtc can have an identical zpos value. The rule to solving the conflict is to compare the plane object IDs; the plane with a higher ID must be stacked on top of a plane with a lower ID.

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 call *drm_atomic_normalize_zpos()* to update this before it can be trusted.

src clipped source coordinates of the plane (in 16.16)

dst clipped destination coordinates of the plane

visible Visibility of the plane. This can be false even if fb!=NULL and crtc!=NULL, due to clipping.

state backpointer to global drm_atomic_state

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);
};
```

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 paramters 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 -<errorname>EINVAL</errorname> 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_SETPLANE 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.

Drivers implementing atomic modeset should use *drm_atomic_helper_plane_set_property()* to implement this hook.

This callback is optional if the driver does not support any legacy driver-private properties.

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.

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

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 torndown.

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.

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 Primary planes represent a "main" plane for a CRTC. Primary planes are the planes operated upon by CRTC modesetting and flipping operations described in the *drm_crtc_funcs.page_flip* and *drm_crtc_funcs.set_config* hooks.

DRM_PLANE_TYPE_CURSOR Cursor planes represent a "cursor" plane for a CRTC. Cursor planes are the planes operated upon by the DRM_IOCTL_MODE_CURSOR and DRM_IOCTL_MODE_CURSOR2 IOCTLs.

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 (beyong 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()*.

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;
  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 * zpos_property;
  struct drm_property * rotation_property;
};
```

Members

dev DRM device this plane belongs to

head for list management

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

format_types array of formats supported by this plane

format_count number of formats supported

format_default driver hasn't supplied supported formats for the plane

crtc currently bound CRTC

fb currently bound fb

old_fb Temporary tracking of the old fb while a modeset is ongoing. Used by *drm_mode_set_config_internal()* to implement correct refcounting.

funcs helper functions

properties property tracking for this plane

type type of plane (overlay, primary, cursor)

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_plane_in_state()*, *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*.

zpos_property zpos property for this plane

rotation_property rotation property for this plane

unsigned int **drm_plane_index**(struct *drm_plane* * *plane*)
find the index of a registered plane

Parameters

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.

struct *drm_plane* * **drm_plane_find**(struct drm_device * *dev*, uint32_t *id*)
find a *drm_plane*

Parameters

struct drm_device * dev DRM device

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(*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(*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(*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*, 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**

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**.

Return

Zero on success, error code on failure.

int **drm_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*, bool *is_primary*)
Initialize a legacy plane

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**

bool is_primary plane type (primary vs overlay)

Description

Legacy API to initialize a DRM plane.

New drivers should call *drm_universal_plane_init()* instead.

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.

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 horizontal 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_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.

struct **drm_display_mode**
DRM kernel-internal display mode structure

Definition

```
struct drm_display_mode {
  struct list_head head;
  struct drm_mode_object base;
  char name;
  enum drm_mode_status status;
  unsigned int type;
  int clock;
  int hdisplay;
  int hsync_start;
  int hsync_end;
  int htotal;
  int hskew;
  int vdisplay;
  int vsync_start;
  int vsync_end;
  int vtotal;
  int vscan;
  unsigned int flags;
  int width_mm;
  int height_mm;
  int crtc_clock;
  int crtc_hdisplay;
  int crtc_hblank_start;
  int crtc_hblank_end;
  int crtc_hsync_start;
  int crtc_hsync_end;
  int crtc_htotal;
  int crtc_hskew;
  int crtc_vdisplay;
  int crtc_vblank_start;
  int crtc_vblank_end;
  int crtc_vsync_start;
  int crtc_vsync_end;
  int crtc_vtotal;
  int * private;
  int private_flags;
  int vrefresh;
  int hsync;
```

```
  enum hdmi_picture_aspect picture_aspect_ratio;
};
```

Members

head struct list_head for mode lists.

base A display mode is a normal modeset object, possibly including public userspace id.

FIXME:

This can probably be removed since the entire concept of userspace managing modes explicitly has never landed in upstream kernel mode setting support.

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*.

type A bitmask of flags, mostly about the source of a mode. Possible flags are:

- DRM_MODE_TYPE_BUILTIN: Meant for hard-coded modes, effectively unused.
- 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.

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:

- 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.
- DRM_MODE_TYPE_USERDEF: Mode defined by userspace, again a vestige from older kms designs where userspace had to first add a custom mode to the kernel's mode list before it could use it. Don't use.

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: not used?
- DRM_MODE_FLAG_PIXMUX: not used?
- 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.

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.

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

private Pointer for driver private data. This can only be used for mode objects passed to drivers in modeset operations. It shouldn't be used by atomic drivers since they can store any additional data by subclassing state structures.

private_flags Similar to **private**, but just an integer.

vrefresh Vertical refresh rate, for debug output in human readable form. Not used in a functional way.

This value is in Hz.

hsync Horizontal refresh rate, for debug output in human readable form. Not used in a functional way.

This value is in kHz.

picture_aspect_ratio Field for setting the HDMI picture aspect ratio of a mode.

Description

The horizontal and vertical timings are defined per the following diagram.

```
          Active                 Front           Sync           Back
         Region                 Porch                          Porch
<-----------------------><----------------><-------------><-------------->
  //////////////////////|
 ////////////////////// |
//////////////////////  |..................               ................
                                           _______________
<----- [hv]display ----->
<------------- [hv]sync_start ------------>
<--------------------- [hv]sync_end --------------------->
<-------------------------------- [hv]total ----------------------------->*
```

This structure contains two copies of timings. First are the plain timings, which specify the logical mode, as it would be for a progressive 1:1 scanout at the refresh rate userspace can observe through vblank timestamps. Then there's the hardware timings, which are corrected for interlacing, double-clocking and similar things. They are provided as a convenience, and can be appropriately computed using *drm_mode_set_crtcinfo()*.

For printing you can use DRM_MODE_FMT and *DRM_MODE_ARG()*.

DRM_MODE_FMT()
printf string for *struct drm_display_mode*

Parameters

DRM_MODE_ARG(*m*)
printf arguments for *struct drm_display_mode*

Parameters

m display mode

bool **drm_mode_is_stereo**(const struct *drm_display_mode* * *mode*)
check for stereo mode flags

Parameters

const struct drm_display_mode * mode drm_display_mode to check

Return

True if the mode is one of the stereo modes (like side-by-side), false if not.

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_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 http://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 and DE polarity will be stored here

Description

Sets DRM_BUS_FLAG_DE_(LOW|HIGH) and DRM_BUS_FLAG_PIXDATA_(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 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.

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_hsync**(const struct *drm_display_mode* * *mode*)
get the hsync of a mode

Parameters

const struct drm_display_mode * mode mode

Return

modes's hsync rate in kHz, rounded to the nearest integer. Calculates the value first if it is not yet set.

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 object id and list head of the destination mode.

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_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_basic**(const struct *drm_display_mode* * *mode*)
make sure the mode is somewhat sane

Parameters

const struct drm_display_mode * mode mode to check

Description

Check that the mode timings are at least somewhat reasonable. Any hardware specific limits are left up for each driver to check.

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

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 who's 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_mode_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*, 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

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. If **mode_option** is NULL the default command line modeline in fb_mode_option will be parsed instead.

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]

The intermediate drm_cmdline_mode structure is required to store additional options from the command line modline 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.

4.11 Connector Abstraction

In DRM connectors are the general abstraction for display sinks, and include als 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_mode_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 realiable 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!

struct **drm_scrambling**

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_info**
    runtime information about the connected HDMI sink

Definition

```
struct drm_hdmi_info {
  struct drm_scdc scdc;
};
```

Members

scdc sink's scdc support and capabilities

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.

struct **drm_display_info**
runtime data about the connected sink

Definition

```
struct drm_display_info {
  char name;
  unsigned int width_mm;
  unsigned int height_mm;
  unsigned int pixel_clock;
  unsigned int bpc;
  enum subpixel_order subpixel_order;
#define DRM_COLOR_FORMAT_RGB444               (1\\\lt;\\\lt;0
#define DRM_COLOR_FORMAT_YCRCB444     (1\\\lt;\\\lt;1
#define DRM_COLOR_FORMAT_YCRCB422     (1\\\lt;\\\lt;2
  u32 color_formats;
  const u32 * bus_formats;
  unsigned int num_bus_formats;
#define DRM_BUS_FLAG_DE_LOW           (1\\\lt;\\\lt;0
#define DRM_BUS_FLAG_DE_HIGH          (1\\\lt;\\\lt;1
#define DRM_BUS_FLAG_PIXDATA_POSEDGE  (1\\\lt;\\\lt;2
#define DRM_BUS_FLAG_PIXDATA_NEGEDGE  (1\\\lt;\\\lt;3
#define DRM_BUS_FLAG_DATA_MSB_TO_LSB  (1\\\lt;\\\lt;4
#define DRM_BUS_FLAG_DATA_LSB_TO_MSB  (1\\\lt;\\\lt;5
  u32 bus_flags;
  int max_tmds_clock;
  bool dvi_dual;
  u8 edid_hdmi_dc_modes;
  u8 cea_rev;
  struct drm_hdmi_info hdmi;
};
```

Members

name Name of the display.

width_mm Physical width in mm.

height_mm Physical height in mm.

pixel_clock Maximum pixel clock supported by the sink, in units of 100Hz. This mismatches the clock in *drm_display_mode* (which is in kHZ), because that's what the EDID uses as base unit.

bpc Maximum bits per color channel. Used by HDMI and DP outputs.

subpixel_order Subpixel order of LCD panels.

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 DRM_BUS_FLAGS_ defines.

max_tmds_clock Maximum TMDS clock rate supported by the sink in kHz. 0 means undefined.

dvi_dual Dual-link DVI sink?

edid_hdmi_dc_modes Mask of supported hdmi deep color modes. Even more stuff redundant with **bus_formats**.

cea_rev CEA revision of the HDMI sink.

hdmi advance features of a HDMI sink.

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_tv_connector_state**
TV connector related states

Definition

```
struct drm_tv_connector_state {
  enum drm_mode_subconnector subconnector;
  struct margins;
  unsigned int mode;
  unsigned int brightness;
  unsigned int contrast;
  unsigned int flicker_reduction;
  unsigned int overscan;
  unsigned int saturation;
  unsigned int hue;
};
```

Members

subconnector selected subconnector

margins left/right/top/bottom margins

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_tv_connector_state tv;
  enum hdmi_picture_aspect picture_aspect_ratio;
  unsigned int scaling_mode;
};
```

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 can be used by helpers and drivers to select the encoder

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

tv TV connector state

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 `hdmi_picture_aspect`

scaling_mode Connector property to control the upscaling, mostly used for built-in panels.

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);
};
```

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.

Drivers implementing atomic modeset should use *drm_atomic_helper_connector_dpms()* to implement this hook.

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.

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()* or `drm_helper_probe_single_connector_modes_nomerge()` 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.

Drivers implementing atomic modeset should use *drm_atomic_helper_connector_set_property()* to implement this hook.

This callback is optional if the driver does not support any legacy driver-private properties.

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.

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

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.

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_connector`**
central DRM connector control structure

Definition

```
struct drm_connector {
  struct drm_device * dev;
  struct device * kdev;
  struct device_attribute * attr;
  struct list_head head;
  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 registered;
  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_blob * path_blob_ptr;
  struct drm_property_blob * tile_blob_ptr;
#define DRM_CONNECTOR_POLL_HPD (1 \\\lt;\\\lt; 0
```

```
#define DRM_CONNECTOR_POLL_CONNECT (1 \\\lt;\\\lt; 1
#define DRM_CONNECTOR_POLL_DISCONNECT (1 \\\lt;\\\lt; 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;
  bool override_edid;
#define DRM_CONNECTOR_MAX_ENCODER 3
  uint32_t encoder_ids;
  struct drm_encoder * encoder;
#define MAX_ELD_BYTES 128
  uint8_t eld;
  bool latency_present;
  int video_latency;
  int audio_latency;
  int null_edid_counter;
  unsigned bad_edid_counter;
  bool edid_corrupt;
  struct dentry * debugfs_entry;
  struct drm_connector_state * state;
  bool has_tile;
  struct drm_tile_group * tile_group;
  bool tile_is_single_monitor;
  uint8_t num_h_tile;
  uint8_t num_v_tile;
  uint8_t tile_h_loc;
  uint8_t tile_v_loc;
  uint16_t tile_h_size;
  uint16_t tile_v_size;
};
```

Members

dev parent DRM device

kdev kernel device for sysfs attributes

attr sysfs attributes

head list management

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?

doublescan_allowed can this connector handle doublescan?

stereo_allowed can this connector handle stereo modes?

registered Is this connector exposed (registered) with userspace? 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

properties property tracking for this connector

scaling_mode_property Optional atomic property to control the upscaling.

path_blob_ptr DRM blob property data for the DP MST path property.

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.

polled Connector polling mode, a combination of

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

DRM_CONNECTOR_POLL_CONNECT Periodically poll the connector for connection.

DRM_CONNECTOR_POLL_DISCONNECT Periodically poll the connector for disconnection.

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

dpms current dpms state

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

override_edid has the EDID been overwritten through debugfs for testing?

encoder_ids valid encoders for this connector

encoder encoder driving this connector, if any

eld EDID-like data, if present

latency_present AV delay info from ELD, if found

video_latency video latency info from ELD, if found

audio_latency audio latency info from ELD, if found

null_edid_counter track sinks that give us all zeros for the EDID

bad_edid_counter track sinks that give us an EDID with invalid checksum

edid_corrupt indicates whether the last read EDID was corrupt

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_connector_in_state()*, *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*.

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.

Description

Each connector may be connected to one or more CRTCs, 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*, uint32_t *id*)
lookup connector object

Parameters

struct drm_device * dev DRM device

uint32_t id connector object id

Description

This function looks up the connector object specified by id add 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.

void **drm_connector_reference**(struct *drm_connector* * *connector*)
acquire a connector reference

Parameters

struct drm_connector * connector DRM connector

Description

This is a compatibility alias for *drm_connector_get()* and should not be used by new code.

void **drm_connector_unreference**(struct *drm_connector* * *connector*)
release a connector reference

Parameters

struct drm_connector * connector DRM connector

Description

This is a compatibility alias for *drm_connector_put()* and should not be used by new code.

struct **drm_tile_group**
Tile group metadata

Definition

```
struct drm_tile_group {
  struct kref refcount;
  struct drm_device * dev;
  int id;
  u8 group_data;
};
```

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()*.

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()*.

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.

Return

Zero on success, error code on failure.

int **drm_mode_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.

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

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

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.

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 connectr_list iterator

Description

Returns 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.

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.

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.

int **drm_mode_create_tv_properties**(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.

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.picture_aspect_ratio* in the atomic state.

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.picture_aspect_ratio* 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_suggested_offset_properties**(struct drm_device * *dev*)
create suggests offset properties

Parameters

struct drm_device * dev DRM device

Description

Create the the suggested x/y offset property for connectors.

int **drm_mode_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_mode_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.

Return

Zero on success, errno on failure.

int **drm_mode_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.

Return

Zero on success, negative errno on failure.

void **drm_mode_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.

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.

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.

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*, char *topology*)
get a reference to an existing tile group

Parameters

struct drm_device * dev DRM device

char topology 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*, char *topology*)
create a tile group from a displayid description

Parameters

struct drm_device * dev DRM device

char topology 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 error.

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 torndown.

Description

Encoders sit between CRTCs 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 drm_bridge * bridge;
  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 mutliple 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_encoder_init()*.

In reality almost every driver gets this wrong.

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_encoder_init()*. 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 another bits set.

In reality almost every driver gets this wrong.

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

bridge bridge associated to the encoder

funcs control functions

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.

unsigned int **drm_encoder_index**(struct *drm_encoder* * *encoder*)
find the index of a registered encoder

Parameters

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.

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*, uint32_t *id*)
find a *drm_encoder*

Parameters

struct drm_device * dev DRM device

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(*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(*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

Initialises a preallocated encoder. Encoder should be subclassed as part of driver encoder objects. At driver unload time *drm_encoder_cleanup()* should be called from the driver's *drm_encoder_funcs.destroy* hook.

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.

4.13 KMS Initialization and Cleanup

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

4.13.1 CRTCs (struct drm_crtc)

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

CRTC Initialization

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

4.13.2 Cleanup

The DRM core manages its objects' lifetime. When an object is not needed anymore the core calls its destroy function, which must clean up and free every resource allocated for the object. Every `drm_*_init()` call must be matched with a corresponding `drm_*_cleanup()` call to cleanup CRTCs (*drm_crtc_cleanup()*), planes (*drm_plane_cleanup()*), encoders (*drm_encoder_cleanup()*) and connectors (*drm_connector_cleanup()*). Furthermore, connectors that have been added to sysfs must be removed by a call to *drm_connector_unregister()* before calling *drm_connector_cleanup()*.

Connectors state change detection must be cleanup up with a call to *drm_kms_helper_poll_fini()*.

4.13.3 Output discovery and initialization example

```
void intel_crt_init(struct drm_device *dev)
{
    struct drm_connector *connector;
    struct intel_output *intel_output;

    intel_output = kzalloc(sizeof(struct intel_output), GFP_KERNEL);
    if (!intel_output)
        return;

    connector = &intel_output->base;
    drm_connector_init(dev, &intel_output->base,
               &intel_crt_connector_funcs, DRM_MODE_CONNECTOR_VGA);

    drm_encoder_init(dev, &intel_output->enc, &intel_crt_enc_funcs,
             DRM_MODE_ENCODER_DAC);

    drm_mode_connector_attach_encoder(&intel_output->base,
                      &intel_output->enc);

    /* Set up the DDC bus. */
    intel_output->ddc_bus = intel_i2c_create(dev, GPIOA, "CRTDDC_A");
    if (!intel_output->ddc_bus) {
        dev_printk(KERN_ERR, &dev->pdev->dev, "DDC bus registration "
               "failed.\n");
        return;
    }

    intel_output->type = INTEL_OUTPUT_ANALOG;
    connector->interlace_allowed = 0;
    connector->doublescan_allowed = 0;
```

```
    drm_encoder_helper_add(&intel_output->enc, &intel_crt_helper_funcs);
    drm_connector_helper_add(connector, &intel_crt_connector_helper_funcs);

    drm_connector_register(connector);
}
```

In the example above (taken from the i915 driver), a CRTC, connector and encoder combination is created. A device-specific i2c bus is also created for fetching EDID data and performing monitor detection. Once the process is complete, the new connector is registered with sysfs to make its properties available to applications.

4.14 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.txt

The basic usage pattern is to:

```
drm_modeset_acquire_init(:c:type:`ctx`)
retry:
foreach (lock in random_ordered_set_of_locks) {
    ret = drm_modeset_lock(lock, :c:type:`ctx`)
    if (ret == -EDEADLK) {
        drm_modeset_backoff(:c:type:`ctx`);
        goto retry;
    }
}
... do stuff ...
drm_modeset_drop_locks(:c:type:`ctx`);
drm_modeset_acquire_fini(:c:type:`ctx`);
```

On top of 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;
  struct list_head locked;
  bool trylock_only;
};
```

Members

`ww_ctx` base acquire ctx

`contended` used internally for -EDEADLK handling

`locked` list of held locks

`trylock_only` trylock mode used in atomic contexts/panic notifiers

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 it's place on `drm_atomi_state.locked` list when part of an atomic update

Description

Used for locking CRTCs 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_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 for future

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.

void **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.

int **drm_modeset_backoff_interruptible**(struct *drm_modeset_acquire_ctx* * *ctx*)
deadlock avoidance backoff

Parameters

struct drm_modeset_acquire_ctx * ctx the acquire context

Description

Interruptible version of *drm_modeset_backoff()*

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()*.

int **drm_modeset_lock_interruptible**(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

Interruptible version of *drm_modeset_lock()*

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**.

Return

0 on success or a negative error-code on failure.

4.15 KMS Properties

4.15.1 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 instatiated 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;
};
```

Members

value numeric property value for this enum entry

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;
  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_PROB_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 where userspace 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 usersapce, e.g. the EDID, or the connector path property on DP MST sinks.

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 metadatatransport 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;
  unsigned char 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 becauase the uapi encoding of property types is a bit special for historical reasons.

struct *drm_property_blob* * **drm_property_reference_blob**(struct *drm_property_blob* * *blob*)
acquire a blob property reference

Parameters

struct drm_property_blob * blob DRM blob property

Description

This is a compatibility alias for *drm_property_blob_get()* and should not be used by new code.

void **drm_property_unreference_blob**(struct *drm_property_blob* * *blob*)
release a blob property reference

Parameters

struct drm_property_blob * blob DRM blob property

Description

This is a compatibility alias for *drm_property_blob_put()* and should not be used by new code.

struct *drm_property* * **drm_property_find**(struct drm_device * *dev*, uint32_t *id*)
find property object

Parameters

struct drm_device * dev DRM device

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*, int *flags*, const char * *name*, int *num_values*)
create a new property type

Parameters

struct drm_device * dev drm device

int 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*, int *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

int 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*, int *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

int 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*, int *flags*, const char * *name*, uint64_t *min*, uint64_t *max*)

create a new unsigned ranged property type

Parameters

struct drm_device * dev drm device

int 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*, int *flags*, const char * *name*, int64_t *min*, int64_t *max*)

create a new signed ranged property type

Parameters

struct drm_device * dev drm device

int 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*, int *flags*, const char * *name*, uint32_t *type*)

create a new object property type

Parameters

struct drm_device * dev drm device

int 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*, int *flags*, const char * *name*)

create a new boolean property type

Parameters

struct drm_device * dev drm device

int 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*, int *index*, uint64_t *value*, const char * *name*)

add a possible value to an enumeration property

Parameters

struct drm_property * property enumeration property to change

int index index of the new enumeration

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 destry

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 unreference 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.

4.15.2 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_mode_connector_update_edid_property()*, usually after having parsed the EDID using *drm_add_edid_modes()*. Userspace cannot change this property.

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. Atomic drivers should implement this hook using *drm_atomic_helper_connector_dpms()*. This is the only property standard connector property that userspace can change.

PATH: Connector path property to identify how this sink is physically connected. Used by DP MST. This should be set by calling *drm_mode_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 virtualize both *drm_crtc* and *drm_plane* if needed. Drivers should update this value using *drm_mode_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_mode_connector_set_link_status_property()*.

Connectors also have one standardized atomic property:

CRTC_ID: Mode object ID of the *drm_crtc* this connector should be connected to.

4.15.3 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 apprpriately 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:

- 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.
- 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.

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).

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 enumaration 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_ROTATE_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. Once mutable zpos property has been enabled, the DRM core will automatically calculate *drm_plane_state.normalized_zpos* values. 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), driver should adjust min/max values and assign those planes immutable zpos property with lower or higher values (for more information, see *drm_plane_create_zpos_immutable_property()* function). In such case driver should 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 immuttable 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

4.15.4 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 framebuffer 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": Unsinged 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*.

"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()*. Drivers which support both should use *drm_atomic_helper_legacy_gamma_set()* to alias the legacy gamma ramp with the "GAMMA_LUT" property above.

uint32_t **drm_color_lut_extract**(uint32_t *user_input*, uint32_t *bit_precision*)
clamp and round LUT entries

Parameters

uint32_t user_input input value

uint32_t 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.

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.

4.15.5 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.

4.15.6 Explicit Fencing Properties

Explicit fencing allows userspace to control the buffer synchronization between devices. A Fence or a group of fences are transfered 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. For shared buffers represented with a `struct dma_buf` this is tracked in `struct reservation_object`. 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 set the implicit fence using *drm_atomic_set_fence_for_plane()*, 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.

4.15.7 Existing KMS Properties

The following table gives description of drm properties exposed by various modules/drivers.

Owner Module/Drivers	Group	Property Name	Type	Property Values	Object attached	Descriptio
		"scaling mode"	ENUM	{ "None", "Full", "Center", "Full aspect" }	Connector	Supported by: amdg gma500, i915, nouv and radeon
	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

Continued on next p

Table 4.1 – continued from previous page

Owner Module/Drivers	Group	Property Name	Type	Property Values	Object attached	Descriptio
		"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
		"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 suggest X offset fo connector
		"suggested Y"	RANGE	Min=0, Max=0xffffffff	Connector	property suggest Y offset fo connector
	Optional	"aspect ratio"	ENUM	{ "None", "4:3", "16:9" }	Connector	TDB

Continued on next p

Table 4.1 – continued from previous page

Owner Module/Drivers	Group	Property Name	Type	Property Values	Object attached	Descriptio
i915	Generic	"Broadcast RGB"	ENUM	{ "Automatic", "Full", "Limited 16:235" }	Connector	When property set to L ited 16: and CTM set, the h ware will programm with the sult of multiplicat of CTM the lim range ma to ensure pixels maly in range 0. are remap to the ra 16/255..23
		"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 dependent	Connector	TBD
		"right_margin"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"top_margin"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"bottom_margin"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"hpos"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"vpos"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"contrast"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"saturation"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD

Continued on next p

Table 4.1 – continued from previous page

Owner Module/Drivers	Group	Property Name	Type	Property Values	Object attached	Descriptio
		"hue"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"sharpness"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"flicker_filter"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"flicker_filter_adaptive"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"flicker_filter_2d"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"tv_chroma_filter"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"tv_luma_filter"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"dot_crawl"	RANGE	Min=0, Max=1	Connector	TBD
	SDVO-TV/LVDS	"brightness"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
CDV gma-500	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 dependent	Connector	TBD
		"right_margin"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"top_margin"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"bottom_margin"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"hpos"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD

Continued on next p

Table 4.1 – continued from previous page

Owner Module/Drivers	Group	Property Name	Type	Property Values	Object attached	Descriptio
		"vpos"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"contrast"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"saturation"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"hue"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"sharpness"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"flicker_filter"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"flicker_filter_adaptive"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"flicker_filter_2d"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"tv_chroma_filter"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"tv_luma_filter"	RANGE	Min=0, Max= SDVO dependent	Connector	TBD
		"dot_crawl"	RANGE	Min=0, Max=1	Connector	TBD
	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=0xffffff	Plane	TBD
		"colorkey_min"	RANGE	Min=0, Max=0xffffff	Plane	TBD
		"colorkey_max"	RANGE	Min=0, Max=0xffffff	Plane	TBD
		"colorkey_val"	RANGE	Min=0, Max=0xffffff	Plane	TBD
		"colorkey_alpha"	RANGE	Min=0, Max=0xffffff	Plane	TBD

Continued on next p

Table 4.1 – continued from previous page

Owner Module/Drivers	Group	Property Name	Type	Property Values	Object attached	Descriptio
		"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
exynos	CRTC	"mode"	ENUM	{ "normal", "blank" }	CRTC	TBD
i2c/ch7006_drv	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=0x01ffffff	Plane	TBD
		"contrast"	RANGE	Min=0, Max=8192-1	Plane	TBD
		"brightness"	RANGE	Min=0, Max=1024	Plane	TBD
		"hue"	RANGE	Min=0, Max=359	Plane	TBD
		"saturation"	RANGE	Min=0, Max=8192-1	Plane	TBD
		"iturbt_709"	RANGE	Min=0, Max=1	Plane	TBD
	Nv04 Overlay	"colorkey"	RANGE	Min=0, Max=0x01ffffff	Plane	TBD
		"brightness"	RANGE	Min=0, Max=1024	Plane	TBD
	Display	"dithering mode"	ENUM	{ "auto", "off", "on" }	Connector	TBD
		"dithering depth"	ENUM	{ "auto", "off", "on", "static 2x2", "dynamic 2x2", "temporal" }	Connector	TBD

Continued on next p

Table 4.1 – continued from previous page

Owner Module/Drivers	Group	Property Name	Type	Property Values	Object attached	Descriptio
		"underscan"	ENUM	{ "auto", "6 bpc", "8 bpc" }	Connector	TBD
		"underscan hborder"	RANGE	Min=0, Max=128	Connector	TBD
		"underscan 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	"hotplug_mode_update"	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", "secam" }	Connector	TBD
	legacy TMDS PLL detect	"tmds_pll"	ENUM	{ "driver", "bios" }	•	TBD
	Underscan	"underscan"	ENUM	{ "off", "on", "auto" }	Connector	TBD
		"underscan hborder"	RANGE	Min=0, Max=128	Connector	TBD
		"underscan vborder"	RANGE	Min=0, Max=128	Connector	TBD
	Audio	"audio"	ENUM	{ "off", "on", "auto" }	Connector	TBD
	FMT Dithering	"dither"	ENUM	{ "off", "on" }	Connector	TBD
rcar-du	Generic	"alpha"	RANGE	Min=0, Max=255	Plane	TBD
		"colorkey"	RANGE	Min=0, Max=0x01ffffff	Plane	TBD

4.16 Vertical Blanking

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

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

- int (*enable_vblank) (struct drm_device *dev, int crtc); void (*disable_vblank) (struct drm_device *dev, int crtc); Enable or disable vertical blanking interrupts for the given CRTC.
- u32 (*get_vblank_counter) (struct drm_device *dev, int crtc); Retrieve the value of the vertical blanking counter for the given CRTC. If the hardware maintains a vertical blanking counter its value should be returned. Otherwise drivers can use the `drm_vblank_count()` helper function to handle this operation.

Drivers must initialize the vertical blanking handling core with a call to *`drm_vblank_init()`* in their load operation.

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

To decrement the use count drivers call `drm_vblank_put()`. Only when the use count drops to zero will the DRM core disable the vertical blanking interrupts after a delay by scheduling a timer. The delay is accessible through the vblankoffdelay module parameter or the `drm_vblank_offdelay` global variable and expressed in milliseconds. Its default value is 5000 ms. Zero means never disable, and a negative value means disable immediately. Drivers may override the behaviour by setting the `struct drm_device` vblank_disable_immediate flag, which when set causes vblank interrupts to be disabled immediately regardless of the drm_vblank_offdelay value. The flag should only be set if there's a properly working hardware vblank counter present.

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

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

4.16.1 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;
  struct drm_event_vblank event;
};
```

Members

`base` Base structure for tracking pending DRM events.

`pipe` *`drm_crtc_index()`* of the *`drm_crtc`* this event is for.

`event` Actual event which will be sent to userspace.

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;
  u32 count;
  struct timeval time;
```

```
  atomic_t refcount;
  u32 last;
  unsigned int inmodeset;
  unsigned int pipe;
  int framedur_ns;
  int linedur_ns;
  struct drm_display_mode hwmode;
  bool enabled;
};
```

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.

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.

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_calc_vbltimestamp_from_scanoutpos()* and computed by *drm_calc_timestamping_constants()*.

linedur_ns Line duration in ns, used by *drm_calc_vbltimestamp_from_scanoutpos()* 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_calc_vbltimestamp_from_scanoutpos()*. 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.

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.

u32 **drm_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 irq's.

This is mostly useful for hardware that can obtain the scanout position, but doesn't have a frame counter.

void **drm_vblank_cleanup**(struct drm_device * *dev*)
cleanup vblank support

Parameters

struct drm_device * dev DRM device

Description

This function cleans up any resources allocated in drm_vblank_init.

Drivers which don't use *drm_irq_install()* need to set `drm_device.irq_enabled` themselves, to signal to the DRM core that vblank interrupts are enabled.

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.

Return

Zero on success or a negative error code on failure.

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_calc_vbltimestamp_from_scanoutpos()*. They are derived from CRTC's true scanout timing, so they take things like panel scaling or other adjustments into account.

bool **drm_calc_vbltimestamp_from_scanoutpos**(struct drm_device * *dev*, unsigned int *pipe*, int * *max_error*, struct timeval * *vblank_time*, bool *in_vblank_irq*)
precise vblank timestamp helper

Parameters

struct drm_device * dev DRM device

unsigned int pipe index of CRTC whose vblank timestamp to retrieve

int * max_error Desired maximum allowable error in timestamps (nanosecs) On return contains true maximum error of timestamp

struct timeval * vblank_time Pointer to struct timeval 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 called from within `get_vblank_timestamp()` implementation of a kms driver to implement the actual timestamping.

Should return timestamps conforming to the OML_sync_control OpenML extension specification. The timestamp corresponds to the end of the vblank interval, aka start of scanout of topmost-leftmost display pixel in the following video frame.

Requires support for optional dev->driver->:c:func:*get_scanout_position()* in kms driver, plus a bit of setup code to provide a drm_display_mode that corresponds to the true scanout timing.

The current implementation only handles standard video modes. It returns as no operation if a doublescan or interlaced video mode is active. Higher level code is expected to handle this.

This function can be used to implement the *drm_driver.get_vblank_timestamp* directly, if the driver implements the *drm_driver.get_scanout_position* hook.

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_update_legacy_modeset_state()*.

Return

Returns true on success, and false on failure, i.e. when no accurate timestamp could be acquired.

u32 **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.

Return

The software vblank counter.

u32 **drm_crtc_vblank_count_and_time**(struct *drm_crtc* * *crtc*, struct timeval * *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

struct timeval * vblanktime Pointer to struct timeval 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.

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.

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 again the next vblank interrupt racing with either this function call or the atomic commit operation. A possible sequence could be:

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 event lock. Caller must also hold a vblank reference for the event **e**, which will be dropped when the next vblank arrives.

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.

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.

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.

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. 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.

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()*.

Return

True if the event was successfully handled, false on failure.

CHAPTER

FIVE

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 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 (* load_lut) (struct drm_crtc *crtc);
  void (* disable) (struct drm_crtc *crtc);
  void (* enable) (struct drm_crtc *crtc);
  int (* atomic_check) (struct drm_crtc *crtc, struct drm_crtc_state *state);
  void (* atomic_begin) (struct drm_crtc *crtc, struct drm_crtc_state *old_crtc_state);
  void (* atomic_flush) (struct drm_crtc *crtc, struct drm_crtc_state *old_crtc_state);
  void (* atomic_disable) (struct drm_crtc *crtc, struct drm_crtc_state *old_crtc_state);
};
```

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 **enable** and **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 **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 **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. It is also used by the transitional plane helpers to implement a **mode_set** hook in *drm_helper_crtc_mode_set()*.

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 **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.

load_lut Load a LUT prepared with the *drm_fb_helper_funcs.gamma_set* vfunc.

This callback is optional and is only used by the fbdev emulation helpers.

FIXME:

This callback is functionally redundant with the core gamma table support and simply exists because the fbdev hasn't yet been refactored to use the core gamma table interfaces.

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 **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.

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 **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 **enable** must be the inverse of **disable** for atomic drivers.

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.atomc_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 and by the transitional plane 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 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 and by the transitional plane 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 and by the transitional plane helpers, but it 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.

Comparing to **disable**, this one provides the additional input parameter **old_crtc_state** which could be used to access the old state. Atomic drivers should consider to use this one instead of **disable**.

Description

These hooks are used by the legacy CRTC helpers, the transitional plane 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);
  struct drm_crtc *(* get_crtc) (struct drm_encoder *encoder);
  enum drm_connector_status (* detect) (struct drm_encoder *encoder, struct drm_connector␣
↪*connector);
  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.

get_crtc This callback is used by the legacy CRTC helpers to work around deficiencies in its own book-keeping.

Do not use, use atomic helpers instead, which get the book keeping right.

FIXME:

Currently only nouveau is using this, and as soon as nouveau is atomic we can ditch this hook.

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.

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.

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 used only by atomic helpers, for symmetry with **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, the transitional plane 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);
  struct drm_encoder *(* best_encoder) (struct drm_connector *connector);
  struct drm_encoder *(* atomic_best_encoder) (struct drm_connector *connector, struct drm_
↪connector_state *connector_state);
  int (* atomic_check) (struct drm_connector *connector, struct drm_connector_state *state);
};
```

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_mode_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 driver 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.

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()*.

Finally drivers that support audio probably want to update the ELD data, too, using *drm_edid_to_eld()*.

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()*.

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*.

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 and you are using the atomic helpers. In this case, the core will call *drm_atomic_helper_best_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_atomic_helper_best_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 free-standing state objects passed-in or assembled in the overall *drm_atomic_state* update tracking structure.

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.

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 (* atomic_check) (struct drm_plane *plane, struct drm_plane_state *state);
  void (* atomic_update) (struct drm_plane *plane, struct drm_plane_state *old_state);
  void (* atomic_disable) (struct drm_plane *plane, struct drm_plane_state *old_state);
};
```

Members

prepare_fb This hook is to prepare a framebuffer for scanout by e.g. pinning it's 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 fence member 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.

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 and by the transitional plane helpers, but it is optional.

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 and by the transitional plane helpers, but it is optional.

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 CRTCs, 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 and by the transitional plane 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.

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 and by the transitional plane helpers, but it is optional.

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 and by the transitional plane helpers, but it is optional.

Description

These functions are used by the atomic helpers and by the transitional plane 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);
};
```

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 upate to be executed by the hardware, for example using *drm_atomic_helper_wait_for_vblanks()*, 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()*.

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()*, *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 have to be implemented in the following sequence:

1. Run *drm_atomic_helper_prepare_planes()* first. This is the only function which commit needs to call which can fail, so we want to run it first and synchronously.

2. Synchronize with any outstanding nonblocking commit worker threads which might be affected the new state update. This can be done by either cancelling or flushing the work items, depending upon whether the driver can deal with cancelled updates. Note that it is important to ensure that the framebuffer cleanup is still done when cancelling.

Asynchronous workers need to have sufficient parallelism to be able to run different atomic commits on different CRTCs in parallel. The simplest way to achive 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.

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. And doing

this while it's guaranteed that no relevant nonblocking worker runs means that nonblocking workers do not need grab any locks. Actually they must not grab locks, for otherwise the work flushing will deadlock.

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 above scheme is implemented in the atomic helper libraries in *drm_atomic_helper_commit()* using a bunch of helper functions. See *drm_atomic_helper_setup_commit()* for a starting point.

5.2.3 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, CRTCs 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.

5.2.4 Helper Functions Reference

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(*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(*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_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 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. 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 afterwards 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_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.

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. It also updates the timestamping constants used for precise vblank timestamps by calling *drm_calc_timestamping_constants()*.

Drivers can use this for building their own atomic commit if they don't have a pure helper-based modeset implementation.

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 effected 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.

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.

Note that the default ordering of how the various stages are called is to match the legacy modeset helper library closest. One peculiarity of that is that it doesn't mesh well with runtime PM at all.

For drivers supporting runtime PM the recommended sequence is instead

```
drm_atomic_helper_commit_modeset_disables(dev, old_state);

drm_atomic_helper_commit_modeset_enables(dev, old_state);

drm_atomic_helper_commit_planes(dev, old_state,
                                DRM_PLANE_COMMIT_ACTIVE_ONLY);
```

for committing the atomic update to hardware. See the kerneldoc entries for these three functions for more details.

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 it's 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.

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 it's 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 esay-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.

Return

0 on success. -EBUSY when userspace schedules nonblocking commits too fast, -ENOMEM on allocation failures and -EINTR when a signal is pending.

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_vblank_send_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_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_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 crtcs. 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 CRTCs in one step. 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 CRTCs, 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 savely used when planes are not allowed to move between different CRTCs because this function doesn't handle inter-CRTC depencies. Callers need to ensure that either no such depencies 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 must also be called on the new state when the atomic update fails at any point after calling *drm_atomic_helper_prepare_planes()*.

void **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 proceeding 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 swaped 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.

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_mode_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()*.

Return

0 on success or a negative error code on failure.

See also: *drm_atomic_helper_suspend()*, *drm_atomic_helper_resume()* and *drm_atomic_helper_shutdown()*.

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_crtc_force_disable_all()*.

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.

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.

See also: *drm_atomic_helper_duplicate_state()*, *drm_atomic_helper_disable_all()*, *drm_atomic_helper_resume()*, *drm_atomic_helper_commit_duplicated_state()*

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.

Return

0 on success or a negative error code on failure.

See also: *drm_atomic_helper_suspend()*

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 mode-set 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.

Return

0 on success or a negative error code on failure.

See also: *drm_atomic_helper_suspend()*

int **drm_atomic_helper_crtc_set_property**(struct *drm_crtc* * *crtc*, struct *drm_property* * *property*, uint64_t *val*)

helper for crtc properties

Parameters

struct drm_crtc * crtc DRM crtc

struct drm_property * property DRM property

uint64_t val value of property

Description

Provides a default crtc set_property handler using the atomic driver interface.

Return

Zero on success, error code on failure

int **drm_atomic_helper_plane_set_property**(struct *drm_plane* * *plane*, struct *drm_property* * *property*, uint64_t *val*)

helper for plane properties

Parameters

struct drm_plane * plane DRM plane

struct drm_property * property DRM property

uint64_t val value of property

Description

Provides a default plane set_property handler using the atomic driver interface.

Return

Zero on success, error code on failure

int **drm_atomic_helper_connector_set_property**(struct *drm_connector* * *connector*, struct *drm_property* * *property*, uint64_t *val*)

helper for connector properties

Parameters

struct drm_connector * connector DRM connector

struct drm_property * property DRM property

uint64_t val value of property

Description

Provides a default connector set_property handler using the atomic driver interface.

Return

Zero on success, 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.

Return

Returns 0 on success, negative errno numbers on failure.

See also: `drm_atomic_helper_page_flip_target()`

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.

int **drm_atomic_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

This is the main helper function provided by the atomic helper framework for implementing the legacy DPMS connector interface. It computes the new desired `drm_crtc_state.active` state for the corresponding CRTC (if the connector is enabled) and updates it.

Return

Returns 0 on success, negative errno numbers on failure.

struct *drm_encoder* * **drm_atomic_helper_best_encoder**(struct *drm_connector* * *connector*)
Helper for *drm_connector_helper_funcs.best_encoder* callback

Parameters

struct drm_connector * connector Connector control structure

Description

This is a *drm_connector_helper_funcs.best_encoder* callback helper for connectors that support exactly 1 encoder, statically determined at driver init time.

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_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_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_conector->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_duplicate_state**(struct *drm_connector* * *connector*, struct *drm_connector_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.

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()*.

Return

A pointer to the copy of the atomic state object on success or an ERR_PTR()-encoded error code on failure.

See also: *drm_atomic_helper_suspend()*, *drm_atomic_helper_resume()*

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.

int **drm_atomic_helper_legacy_gamma_set**(struct *drm_crtc* * *crtc*, u16 * *red*, u16 * *green*, u16 * *blue*, uint32_t *size*, struct *drm_modeset_acquire_ctx* * *ctx*)
set the legacy gamma correction table

Parameters

struct drm_crtc * crtc CRTC object

u16 * red red correction table

u16 * green green correction table

u16 * blue green correction table

uint32_t size size of the tables

struct drm_modeset_acquire_ctx * ctx lock acquire context

Description

Implements support for legacy gamma correction table for drivers that support color management through the DEGAMMA_LUT/GAMMA_LUT properties. See *drm_crtc_enable_color_mgmt()* and the containing chapter for how the atomic color management and gamma tables work.

5.3 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.

NOTE

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 ->:c:func:*disable()* hooks on a disabled function, or ->:c:func:*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.

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_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 ->:c:func:*mode_set_base()* helper operation is not provided, the helper function performs a full mode set sequence by calling the ->:c:func:*prepare()*, ->:c:func:*mode_set()* and ->:c:func:*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, for which this function is unsuitable. Instead drivers should use *drm_atomic_helper_connector_dpms()*.

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_crtc_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*)
mode_set implementation for atomic plane helpers

Parameters

struct drm_crtc * crtc DRM CRTC

struct drm_display_mode * mode DRM display mode which userspace requested

struct drm_display_mode * adjusted_mode DRM display mode adjusted by ->mode_fixup callbacks

int x x offset of the CRTC scanout area on the underlying framebuffer

int y y offset of the CRTC scanout area on the underlying framebuffer

struct drm_framebuffer * old_fb previous framebuffer

Description

This function implements a callback useable as the ->mode_set callback required by the CRTC helpers. Besides the atomic plane helper functions for the primary plane the driver must also provide the ->mode_set_nofb callback to set up the CRTC.

This is a transitional helper useful for converting drivers to the atomic interfaces.

int **drm_helper_crtc_mode_set_base**(struct *drm_crtc* * *crtc*, int *x*, int *y*, struct *drm_framebuffer* * *old_fb*)
mode_set_base implementation for atomic plane helpers

Parameters

struct drm_crtc * crtc DRM CRTC

int x x offset of the CRTC scanout area on the underlying framebuffer

int y y offset of the CRTC scanout area on the underlying framebuffer

struct drm_framebuffer * old_fb previous framebuffer

Description

This function implements a callback useable as the ->mode_set_base used required by the CRTC helpers. The driver must provide the atomic plane helper functions for the primary plane.

This is a transitional helper useful for converting drivers to the atomic interfaces.

5.4 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.

struct **drm_simple_display_pipe_funcs**
helper operations for a simple display pipeline

Definition

```
struct drm_simple_display_pipe_funcs {
  void (* enable) (struct drm_simple_display_pipe *pipe, struct drm_crtc_state *crtc_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);
};
```

Members

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()* directly in case the hardware lacks vblank support entirely.

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.

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.

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()*.

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*, 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**

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.

5.5 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.

Initialization is done as a four-step process with *drm_fb_helper_prepare()*, *drm_fb_helper_init()*, *drm_fb_helper_single_add_all_connectors()* and *drm_fb_helper_initial_config()*. Drivers with fancier requirements than the default behaviour can override the third step with their own code. Teardown is done with *drm_fb_helper_fini()* after the fbdev device is unregisters using *drm_fb_helper_unregister_fbi()*.

At runtime drivers should restore the fbdev console by calling *drm_fb_helper_restore_fbdev_mode_unlocked()* from their *drm_driver.lastclose* callback. They should also notify the fb helper code from updates to the output configuration by calling *drm_fb_helper_hotplug_event()*. For easier integration with the output polling code in drm_crtc_helper.c the modeset code provides a *drm_mode_config_funcs.output_poll_changed* callback.

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 CRTCs, 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_single_add_all_connectors()* followed by *drm_fb_helper_initial_config()*.

If *drm_framebuffer_funcs.dirty* is set, the drm_fb_helper_{cfb,sys}_{write,fillrect,copyarea,imageblit} 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.

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 CRTCs 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.

So what is passed to *drm_fb_helper_fill_var()* should be fb_width/fb_height, rather than the surface size.

struct **drm_fb_helper_funcs**
driver callbacks for the fbdev emulation library

Definition

```
struct drm_fb_helper_funcs {
  void (* gamma_set) (struct drm_crtc *crtc, u16 red, u16 green, u16 blue, int regno);
  void (* gamma_get) (struct drm_crtc *crtc, u16 *red, u16 *green, u16 *blue, int regno);
  int (* fb_probe) (struct drm_fb_helper *helper, struct drm_fb_helper_surface_size *sizes);
  bool (* initial_config) (struct drm_fb_helper *fb_helper,struct drm_fb_helper_crtc **crtcs,
 ↪struct drm_display_mode **modes,struct drm_fb_offset *offsets, bool *enabled, int width, int␣
 ↪height);
};
```

Members

gamma_set Set the given gamma LUT register on the given CRTC.

This callback is optional.

FIXME:

This callback is functionally redundant with the core gamma table support and simply exists because the fbdev hasn't yet been refactored to use the core gamma table interfaces.

gamma_get Read the given gamma LUT register on the given CRTC, used to save the current LUT when force-restoring the fbdev for e.g. kdbg.

This callback is optional.

FIXME:

This callback is functionally redundant with the core gamma table support and simply exists because the fbdev hasn't yet been refactored to use the core gamma table interfaces.

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.

initial_config Driver callback to setup an initial fbdev display configuration. Drivers can use this callback to tell the fbdev emulation what the preferred initial configuration is. This is useful to implement smooth booting where the fbdev (and subsequently all userspace) never changes the mode, but always inherits the existing configuration.

This callback is optional.

RETURNS:

The driver should return true if a suitable initial configuration has been filled out and false when the fbdev helper should fall back to the default probing logic.

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_framebuffer * fb;
  struct drm_device * dev;
  int crtc_count;
  struct drm_fb_helper_crtc * crtc_info;
  int connector_count;
  int connector_info_alloc_count;
  struct drm_fb_helper_connector ** connector_info;
  const struct drm_fb_helper_funcs * funcs;
  struct fb_info * fbdev;
  u32 pseudo_palette;
  struct drm_clip_rect dirty_clip;
  spinlock_t dirty_lock;
  struct work_struct dirty_work;
  struct work_struct resume_work;
  struct list_head kernel_fb_list;
  bool delayed_hotplug;
};
```

Members

fb Scanout framebuffer object

dev DRM device

crtc_count number of possible CRTCs

crtc_info per-CRTC helper state (mode, x/y offset, etc)

connector_count number of connected connectors

connector_info_alloc_count size of connector_info

connector_info array of per-connector information

funcs driver callbacks for fb helper

fbdev emulated fbdev device info struct

pseudo_palette fake palette of 16 colors

dirty_clip clip rectangle used with deferred_io to accumulate damage to the screen buffer

dirty_lock spinlock protecting **dirty_clip**

dirty_work worker used to flush the framebuffer

resume_work worker used during resume if the console lock is already taken

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.

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()
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_single_add_all_connectors**(struct *drm_fb_helper* * *fb_helper*)
add all connectors to fbdev emulation helper

Parameters

struct drm_fb_helper * fb_helper fbdev initialized with drm_fb_helper_init

Description

This functions adds all the available connectors for use with the given fb_helper. This is a separate step to allow drivers to freely assign connectors to the fbdev, e.g. if some are reserved for special purposes or not adequate to be used for the fbcon.

This function is protected against concurrent connector hotadds/removals using `drm_fb_helper_add_one_connector()` and `drm_fb_helper_remove_one_connector()`.

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 fbcon to restore

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*, 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

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.

int **drm_fb_helper_init**(struct drm_device * *dev*, struct *drm_fb_helper* * *fb_helper*, int *max_conn_count*)
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

int max_conn_count max connector count

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_fbi**(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 members cmap and apertures. 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_unregister_fbi**(struct *drm_fb_helper* * *fb_helper*)
unregister fb_info framebuffer device

Parameters

struct drm_fb_helper * fb_helper driver-allocated fbdev helper

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*)
finialize a *struct drm_fb_helper*

Parameters

struct drm_fb_helper * fb_helper driver-allocated fbdev helper

Description

This cleans up all remaining resources associated with **fb_helper**. Must be called after *drm_fb_helper_unlink_fbi()* was called.

void **drm_fb_helper_unlink_fbi**(struct *drm_fb_helper* * *fb_helper*)
wrapper around unlink_framebuffer

Parameters

struct drm_fb_helper * fb_helper driver-allocated fbdev helper

Description

A wrapper around unlink_framebuffer implemented by fbdev core

void **drm_fb_helper_deferred_io**(struct fb_info * *info*, struct list_head * *pagelist*)
fbdev deferred_io callback function

Parameters

struct fb_info * info fb_info struct pointer

struct list_head * pagelist list of dirty mmap framebuffer pages

Description

This function is used as the `fb_deferred_io.deferred_io` callback function for flushing the fbdev mmap writes.

ssize_t **drm_fb_helper_sys_read**(struct fb_info * *info*, char __user * *buf*, size_t *count*, loff_t * *ppos*)
wrapper around fb_sys_read

Parameters

struct fb_info * info fb_info struct pointer

char __user * buf userspace buffer to read from framebuffer memory

size_t count number of bytes to read from framebuffer memory

loff_t * ppos read offset within framebuffer memory

Description

A wrapper around fb_sys_read implemented by fbdev core

ssize_t **drm_fb_helper_sys_write**(struct fb_info * *info*, const char __user * *buf*, size_t *count*, loff_t * *ppos*)
wrapper around fb_sys_write

Parameters

struct fb_info * info fb_info struct pointer

const char __user * buf userspace buffer to write to framebuffer memory

size_t count number of bytes to write to framebuffer memory

loff_t * ppos write offset within framebuffer memory

Description

A wrapper around fb_sys_write implemented by fbdev core

void **drm_fb_helper_sys_fillrect**(struct fb_info * *info*, const struct fb_fillrect * *rect*)
wrapper around sys_fillrect

Parameters

struct fb_info * info fbdev registered by the helper

const struct fb_fillrect * rect info about rectangle to fill

Description

A wrapper around sys_fillrect implemented by fbdev core

void **drm_fb_helper_sys_copyarea**(struct fb_info * *info*, const struct fb_copyarea * *area*)
wrapper around sys_copyarea

Parameters

struct fb_info * info fbdev registered by the helper

const struct fb_copyarea * area info about area to copy

Description

A wrapper around sys_copyarea implemented by fbdev core

void **drm_fb_helper_sys_imageblit**(struct fb_info * *info*, const struct fb_image * *image*)
wrapper around sys_imageblit

Parameters

struct fb_info * info fbdev registered by the helper

const struct fb_image * image info about image to blit

Description

A wrapper around sys_imageblit implemented by fbdev core

void **drm_fb_helper_cfb_fillrect**(struct fb_info * *info*, const struct fb_fillrect * *rect*)
wrapper around cfb_fillrect

Parameters

struct fb_info * info fbdev registered by the helper

const struct fb_fillrect * rect info about rectangle to fill

Description

A wrapper around cfb_imageblit implemented by fbdev core

void **drm_fb_helper_cfb_copyarea**(struct fb_info * *info*, const struct fb_copyarea * *area*)
wrapper around cfb_copyarea

Parameters

struct fb_info * info fbdev registered by the helper

const struct fb_copyarea * area info about area to copy

Description

A wrapper around cfb_copyarea implemented by fbdev core

void **drm_fb_helper_cfb_imageblit**(struct fb_info * *info*, const struct fb_image * *image*)
wrapper around cfb_imageblit

Parameters

struct fb_info * info fbdev registered by the helper

const struct fb_image * image info about image to blit

Description

A wrapper around cfb_imageblit implemented by fbdev core

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

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

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_fix**(struct fb_info * *info*, uint32_t *pitch*, uint32_t *depth*)
initializes fixed fbdev information

Parameters

struct fb_info * info fbdev registered by the helper

uint32_t pitch desired pitch

uint32_t depth desired depth

Description

Helper to fill in the fixed fbdev information useful for a non-accelerated fbdev emulations. Drivers which support acceleration methods which impose additional constraints need to set up their own limits.

Drivers should call this (or their equivalent setup code) from their *`drm_fb_helper_funcs.fb_probe`* callback.

void **drm_fb_helper_fill_var**(struct fb_info * *info*, struct *drm_fb_helper* * *fb_helper*, uint32_t *fb_width*, uint32_t *fb_height*)
initalizes variable 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

uint32_t fb_width desired fb width

uint32_t fb_height desired fb height

Description

Sets up the variable 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*, int *bpp_sel*)
setup a sane initial connector configuration

Parameters

struct drm_fb_helper * fb_helper fb_helper device struct

int bpp_sel bpp value to use for the framebuffer configuration

Description

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

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_var()* and *drm_fb_helper_fill_fix()* are provided as helpers 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 the drm_fb_helper

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.

5.6 Framebuffer CMA Helper Functions Reference

Provides helper functions for creating a cma (contiguous memory allocator) backed framebuffer.

drm_fb_cma_create() is used in the *drm_mode_config_funcs.fb_create* callback function to create a cma backed framebuffer.

An fbdev framebuffer backed by cma is also available by calling *drm_fbdev_cma_init()*. *drm_fbdev_cma_fini()* tears it down. If the *drm_framebuffer_funcs.dirty* callback is set, fb_deferred_io will be set up automatically. *drm_framebuffer_funcs.dirty* is called by *drm_fb_helper_deferred_io()* in process context (`struct delayed_work`).

Example fbdev deferred io code:

```
static int driver_fb_dirty(struct drm_framebuffer *fb,
                           struct drm_file *file_priv,
                           unsigned flags, unsigned color,
                           struct drm_clip_rect *clips,
                           unsigned num_clips)
{
    struct drm_gem_cma_object *cma = drm_fb_cma_get_gem_obj(fb, 0);
    ... push changes ...
    return 0;
}

static struct drm_framebuffer_funcs driver_fb_funcs = {
    .destroy       = drm_fb_cma_destroy,
    .create_handle = drm_fb_cma_create_handle,
    .dirty         = driver_fb_dirty,
};
```

Initialize:

```
fbdev = drm_fbdev_cma_init_with_funcs(dev, 16,
                                      dev->mode_config.num_crtc,
                                      dev->mode_config.num_connector,
                                      :c:type:`driver_fb_funcs`);
```

struct *drm_framebuffer* * **drm_fb_cma_create_with_funcs**(struct drm_device * *dev*, struct *drm_file* * *file_priv*, 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_priv drm file for the ioctl call

const struct drm_mode_fb_cmd2 * mode_cmd metadata from the userspace fb creation request

const struct drm_framebuffer_funcs * funcs vtable to be used for the new framebuffer object

Description

This can be used to set *drm_framebuffer_funcs* for drivers that need the *drm_framebuffer_funcs.dirty* callback. Use *drm_fb_cma_create()* if you don't need to change *drm_framebuffer_funcs*.

struct *drm_framebuffer* * **drm_fb_cma_create**(struct drm_device * *dev*, struct *drm_file* * *file_priv*, const struct drm_mode_fb_cmd2 * *mode_cmd*)
drm_mode_config_funcs.fb_create callback function

Parameters

struct drm_device * dev DRM device

struct drm_file * file_priv drm file for the ioctl call

const struct drm_mode_fb_cmd2 * mode_cmd metadata from the userspace fb creation request

Description

If your hardware has special alignment or pitch requirements these should be checked before calling this function. Use *drm_fb_cma_create_with_funcs()* if you need to set *drm_framebuffer_funcs.dirty*.

struct *drm_gem_cma_object* * **drm_fb_cma_get_gem_obj**(struct *drm_framebuffer* * *fb*, unsigned int *plane*)
Get CMA GEM object for framebuffer

Parameters

struct drm_framebuffer * fb The framebuffer

unsigned int plane Which plane

Description

Return the CMA GEM object for given framebuffer.

This function will usually be called from the CRTC callback functions.

dma_addr_t **drm_fb_cma_get_gem_addr**(struct *drm_framebuffer* * *fb*, struct *drm_plane_state* * *state*, unsigned int *plane*)
Get physical address for framebuffer

Parameters

struct drm_framebuffer * fb The framebuffer

struct drm_plane_state * state Which state of drm plane

unsigned int plane Which plane Return the CMA GEM address for given framebuffer.

Description

This function will usually be called from the PLANE callback functions.

int **drm_fb_cma_prepare_fb**(struct *drm_plane* * *plane*, struct *drm_plane_state* * *state*)
Prepare CMA framebuffer

Parameters

struct drm_plane * plane Which plane

struct drm_plane_state * state Plane state attach fence to

Description

This should be set as the *struct drm_plane_helper_funcs*.prepare_fb hook.

This function checks if the plane FB has an dma-buf attached, extracts the exclusive fence and attaches it to plane state for the atomic helper to wait on.

There is no need for cleanup_fb for CMA based framebuffer drivers.

int **drm_fb_cma_debugfs_show**(struct seq_file * *m*, void * *arg*)
Helper to list CMA framebuffer objects in debugfs.

Parameters

struct seq_file * m output file

void * arg private data for the callback

struct drm_fbdev_cma * **drm_fbdev_cma_init_with_funcs**(struct drm_device * *dev*, unsigned int *preferred_bpp*, unsigned int *max_conn_count*, const struct *drm_framebuffer_funcs* * *funcs*)

Allocate and initializes a drm_fbdev_cma struct

Parameters

struct drm_device * dev DRM device

unsigned int preferred_bpp Preferred bits per pixel for the device

unsigned int max_conn_count Maximum number of connectors

const struct drm_framebuffer_funcs * funcs fb helper functions, in particular a custom `dirty()` callback

Description

Returns a newly allocated drm_fbdev_cma struct or a ERR_PTR.

struct drm_fbdev_cma * **drm_fbdev_cma_init**(struct drm_device * *dev*, unsigned int *preferred_bpp*, unsigned int *max_conn_count*)

Allocate and initializes a drm_fbdev_cma struct

Parameters

struct drm_device * dev DRM device

unsigned int preferred_bpp Preferred bits per pixel for the device

unsigned int max_conn_count Maximum number of connectors

Description

Returns a newly allocated drm_fbdev_cma struct or a ERR_PTR.

void **drm_fbdev_cma_fini**(struct drm_fbdev_cma * *fbdev_cma*)

Free drm_fbdev_cma struct

Parameters

struct drm_fbdev_cma * fbdev_cma The drm_fbdev_cma struct

void **drm_fbdev_cma_restore_mode**(struct drm_fbdev_cma * *fbdev_cma*)

Restores initial framebuffer mode

Parameters

struct drm_fbdev_cma * fbdev_cma The drm_fbdev_cma struct, may be NULL

Description

This function is usually called from the *`drm_driver.lastclose`* callback.

void **drm_fbdev_cma_hotplug_event**(struct drm_fbdev_cma * *fbdev_cma*)

Poll for hotpulug events

Parameters

struct drm_fbdev_cma * fbdev_cma The drm_fbdev_cma struct, may be NULL

Description

This function is usually called from the *`drm_mode_config.output_poll_changed`* callback.

void **drm_fbdev_cma_set_suspend**(struct drm_fbdev_cma * *fbdev_cma*, int *state*)

wrapper around drm_fb_helper_set_suspend

Parameters

struct drm_fbdev_cma * fbdev_cma The drm_fbdev_cma struct, may be NULL

int state desired state, zero to resume, non-zero to suspend

Description

Calls drm_fb_helper_set_suspend, which is a wrapper around fb_set_suspend implemented by fbdev core.

void **drm_fbdev_cma_set_suspend_unlocked**(struct drm_fbdev_cma * *fbdev_cma*, int *state*)
wrapper around drm_fb_helper_set_suspend_unlocked

Parameters

struct drm_fbdev_cma * fbdev_cma The drm_fbdev_cma struct, may be NULL

int state desired state, zero to resume, non-zero to suspend

Description

Calls drm_fb_helper_set_suspend, which is a wrapper around fb_set_suspend implemented by fbdev core.

5.7 Bridges

5.7.1 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 an intermediate bridge:

```
encoder ---> bridge B ---> bridge A
```

Here, the output of the encoder feeds to bridge B, and that furthers feeds to bridge A.

The driver using the bridge is responsible to make the associations between the encoder and bridges. Once these links are made, the bridges will participate along with encoder functions to perform mode_set/enable/disable through the ops provided in *drm_bridge_funcs*.

drm_bridge, like drm_panel, aren't drm_mode_object entities like planes, CRTCs, 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.

Bridges can also be chained up using the *drm_bridge.next* pointer.

Both legacy CRTC helpers and the new atomic modeset helpers support bridges.

5.7.2 Default bridge callback sequence

The *drm_bridge_funcs* ops are populated by the bridge driver. The DRM internals (atomic and CRTC helpers) use the helpers defined in drm_bridge.c These helpers call a specific *drm_bridge_funcs* op for all the bridges during encoder configuration.

For detailed specification of the bridge callbacks see *drm_bridge_funcs*.

5.7.3 Bridge Helper Reference

struct **drm_bridge_funcs**
drm_bridge control functions

Definition

```
struct drm_bridge_funcs {
  int (* attach) (struct drm_bridge *bridge);
  void (* detach) (struct drm_bridge *bridge);
  enum drm_mode_status (* mode_valid) (struct drm_bridge *crtc, 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,struct drm_display_mode *mode, struct drm_
↪display_mode *adjusted_mode);
  void (* pre_enable) (struct drm_bridge *bridge);
  void (* enable) (struct drm_bridge *bridge);
};
```

Members

attach This callback is invoked whenever our bridge is being attached to a *drm_encoder*.

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()*.

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**.

RETURNS:

drm_mode_status Enum

mode_fixup This callback is used to validate and adjust a mode. The paramater 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.

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.

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 singals) feeding it is no longer running when this callback is called.

The post_disable callback is optional.

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.

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.

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.

struct **drm_bridge**
central DRM bridge control structure

Definition

```
struct drm_bridge {
  struct drm_device * dev;
  struct drm_encoder * encoder;
  struct drm_bridge * next;
#ifdef CONFIG_OF
  struct device_node * of_node;
#endif
  struct list_head list;
  const struct drm_bridge_funcs * funcs;
  void * driver_private;
};
```

Members

dev DRM device this bridge belongs to

encoder encoder to which this bridge is connected

next the next bridge in the encoder chain

of_node device node pointer to the bridge

list to keep track of all added bridges

funcs control functions

driver_private pointer to the bridge driver's internal context

int **drm_bridge_add**(struct *drm_bridge* * *bridge*)
add the given bridge to the global bridge list

Parameters

struct drm_bridge * bridge bridge control structure

Return

Unconditionally returns Zero.

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*)
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)

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.

Return

Zero on success, error code on failure

bool **drm_bridge_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_mode_valid**(struct *drm_bridge* * *bridge*, 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_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_disable**(struct *drm_bridge* * *bridge*)
disables all bridges in the encoder chain

Parameters

struct drm_bridge * bridge bridge control structure

Description

Calls *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 the encoder's prepare op.

Note

the bridge passed should be the one closest to the encoder

void **drm_bridge_post_disable**(struct *drm_bridge* * *bridge*)
cleans up after disabling all bridges in the encoder chain

Parameters

struct drm_bridge * bridge bridge control structure

Description

Calls *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 the encoder's prepare op.

Note

the bridge passed should be the one closest to the encoder

void **drm_bridge_mode_set**(struct *drm_bridge* * *bridge*, struct *drm_display_mode* * *mode*, struct *drm_display_mode* * *adjusted_mode*)
set proposed mode for all bridges in the encoder chain

Parameters

struct drm_bridge * bridge bridge control structure

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_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_bridge_pre_enable**(struct *drm_bridge* * *bridge*)
prepares for enabling all bridges in the encoder chain

Parameters

struct drm_bridge * bridge bridge control structure

Description

Calls *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 the encoder's commit op.

Note

the bridge passed should be the one closest to the encoder

void **drm_bridge_enable**(struct *drm_bridge* * *bridge*)
enables all bridges in the encoder chain

Parameters

struct drm_bridge * bridge bridge control structure

Description

Calls *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 the encoder's commit op.

Note that the bridge passed should be the one closest to the encoder

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

5.7.4 Panel-Bridge Helper Reference

struct *drm_bridge* * **drm_panel_bridge_add**(struct *drm_panel* * *panel*, u32 *connector_type*)
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.

u32 connector_type The DRM_MODE_CONNECTOR_* for the connector to be created.

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, `drm_bridge_detach()` should be called as normal, then *drm_panel_bridge_remove()* to free it.

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.

5.8 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.

struct **drm_panel_funcs**
perform operations on a given panel

Definition

```
struct drm_panel_funcs {
  int (* disable) (struct drm_panel *panel);
  int (* unprepare) (struct drm_panel *panel);
  int (* prepare) (struct drm_panel *panel);
  int (* enable) (struct drm_panel *panel);
  int (* get_modes) (struct drm_panel *panel);
  int (* get_timings) (struct drm_panel *panel, unsigned int num_timings, struct display_timing␣
 ↪*timings);
};
```

Members

disable disable panel (turn off back light, etc.)

unprepare turn off panel

prepare turn on panel and perform set up

enable enable panel (turn on back light, etc.)

get_modes add modes to the connector that the panel is attached to and return the number of modes added

get_timings copy display timings into the provided array and return the number of display timings available

Description

The .:c:func:*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 .:c:func:*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 .:c:func:*disable()* function. Analogously to .:c:func:*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 .:c:func:*unprepare()* function.

struct **drm_panel**
DRM panel object

Definition

```
struct drm_panel {
  struct drm_device * drm;
  struct drm_connector * connector;
  struct device * dev;
  const struct drm_panel_funcs * funcs;
  struct list_head list;
};
```

Members

drm DRM device owning the panel

connector DRM connector that the panel is attached to

dev parent device of the panel

funcs operations that can be performed on the panel

list panel entry in registry

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_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_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_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_get_modes**(struct *drm_panel* * *panel*)
probe the available display modes of a panel

Parameters

struct drm_panel * panel DRM panel

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 a negative error code on failure.

void **drm_panel_init**(struct *drm_panel* * *panel*)
initialize a panel

Parameters

struct drm_panel * panel DRM panel

Description

Sets up internal fields of the panel so that it can subsequently be added to the registry.

int **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.

Return

0 on success or a negative error code on failure.

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_attach**(struct *drm_panel* * *panel*, struct *drm_connector* * *connector*)
attach a panel to a connector

Parameters

struct drm_panel * panel DRM panel

struct drm_connector * connector DRM connector

Description

After obtaining a pointer to a DRM panel a display driver calls this function to attach a panel to a connector.

An error is returned if the panel is already attached to another connector.

Return

0 on success or a negative error code on failure.

int **drm_panel_detach**(struct *drm_panel* * *panel*)
detach a panel from a connector

Parameters

struct drm_panel * panel DRM panel

Description

Detaches a panel from the connector it is attached to. If a panel is not attached to any connector this is effectively a no-op.

Return

0 on success or a negative error code on failure.

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.

Return

A pointer to the panel registered for the specified device tree node or NULL if no panel matching the device tree node can be found.

5.9 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 .:c:func:*transfer()* implementation. Both native and I2C-over-AUX transactions are supported.

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**
DisplayPort AUX channel

Definition

```
struct drm_dp_aux {
  const char * name;
  struct i2c_adapter ddc;
  struct device * 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);
  unsigned i2c_nack_count;
  unsigned i2c_defer_count;
};
```

Members

name user-visible name of this AUX channel and the I2C-over-AUX adapter

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

crtc backpointer to the crtc that is currently using this AUX channel

hw_mutex internal mutex used for locking transfers

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

i2c_nack_count Counts I2C NACKs, used for DP validation.

i2c_defer_count Counts I2C DEFERs, used for DP validation.

Description

The .dev field should be set to a pointer to the device that implements the AUX channel.

The .name field may be used to specify the name of the I2C adapter. If set to NULL, `dev_name()` of .dev will be used.

Drivers provide a hardware-specific implementation of how transactions are executed via the .:c:func:*transfer()* function. 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 propagate errors from the .:c:func:*transfer()* function, 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.

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 .:c:func:*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.

Note that the aux helper code assumes that the .:c:func:*transfer()* function only modifies the reply field of the drm_dp_aux_msg structure. The retry logic and i2c helpers assume this is the case.

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_LIMITED_M_N The device requires main link attributes Mvid and Nvid to be limited to 16 bits.

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 decriptor 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**.

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

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

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*)
read DPCD link status (bytes 0x202-0x207)

Parameters

struct drm_dp_aux * aux DisplayPort AUX channel

u8 status 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_link_probe**(struct *drm_dp_aux* * *aux*, struct drm_dp_link * *link*)
probe a DisplayPort link for capabilities

Parameters

struct drm_dp_aux * aux DisplayPort AUX channel

struct drm_dp_link * link pointer to structure in which to return link capabilities

Description

The structure filled in by this function can usually be passed directly into *drm_dp_link_power_up()* and *drm_dp_link_configure()* to power up and configure the link based on the link's capabilities.

Returns 0 on success or a negative error code on failure.

int **drm_dp_link_power_up**(struct *drm_dp_aux* * *aux*, struct drm_dp_link * *link*)
power up a DisplayPort link

Parameters

struct drm_dp_aux * aux DisplayPort AUX channel

struct drm_dp_link * link pointer to a structure containing the link configuration

Description

Returns 0 on success or a negative error code on failure.

int **drm_dp_link_power_down**(struct *drm_dp_aux* * *aux*, struct drm_dp_link * *link*)
power down a DisplayPort link

Parameters

struct drm_dp_aux * aux DisplayPort AUX channel

struct drm_dp_link * link pointer to a structure containing the link configuration

Description

Returns 0 on success or a negative error code on failure.

int **drm_dp_link_configure**(struct *drm_dp_aux* * *aux*, struct drm_dp_link * *link*)
configure a DisplayPort link

Parameters

struct drm_dp_aux * aux DisplayPort AUX channel

struct drm_dp_link * link pointer to a structure containing the link configuration

Description

Returns 0 on success or a negative error code on failure.

int **drm_dp_downstream_max_clock**(const u8 *dpcd*, const u8 *port_cap*)
extract branch device max pixel rate for legacy VGA converter or max TMDS clock rate for others

Parameters

const u8 dpcd DisplayPort configuration data

const u8 port_cap port capabilities

Description

Returns max clock in kHz on success or 0 if max clock not defined

int **drm_dp_downstream_max_bpc**(const u8 *dpcd*, const u8 *port_cap*)
extract branch device max bits per component

Parameters

const u8 dpcd DisplayPort configuration data

const u8 port_cap port capabilities

Description

Returns max bpc on success or 0 if max bpc not defined

int **drm_dp_downstream_id**(struct *drm_dp_aux* * *aux*, char *id*)
identify branch device

Parameters

struct drm_dp_aux * aux DisplayPort AUX channel

char id 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*, const u8 *port_cap*, struct *drm_dp_aux* * *aux*)
debug DP branch devices

Parameters

struct seq_file * m pointer for debugfs file

const u8 dpcd DisplayPort configuration data

const u8 port_cap port capabilities

struct drm_dp_aux * aux DisplayPort AUX channel

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. You must still call *drm_dp_aux_register()* once the connector has been registered to allow userspace access to the auxiliary DP channel.

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.

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*)
PSR setup in time usec

Parameters

const u8 psr_cap 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 decriptor 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.

5.10 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 sizo 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 sizo 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**(struct i2c_adapter * *adapter*)
Identify the DP dual mode adaptor

Parameters

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**(enum *drm_dp_dual_mode_type type*, struct i2c_adapter * *adapter*)
Max TMDS clock for DP dual mode adaptor

Parameters

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**(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

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**(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

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. As some type 1 adaptors have problems with registers (see comments in *drm_dp_dual_mode_detect()*) we avoid touching the register, making this function a no-op on type 1 adaptors.

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**(struct i2c_adapter * *adapter*, enum *drm_lspcon_mode* * *mode*)

Parameters

struct i2c_adapter * adapter I2C-over-aux adapter

enum drm_lspcon_mode * mode current lspcon mode of operation output variable

Description

reading offset (0x80, 0x41)

Return

0 on success, sets the current_mode value to appropriate mode -error on failure

int **drm_lspcon_set_mode**(struct i2c_adapter * *adapter*, enum *drm_lspcon_mode mode*)

Parameters

struct i2c_adapter * adapter I2C-over-aux adapter

enum drm_lspcon_mode mode required mode of operation

Description

writing offset (0x80, 0x40)

Return

0 on success, -error on failure/timeout

5.11 Display Port MST Helper Functions Reference

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.

struct **drm_dp_vcpi**
Virtual Channel Payload Identifier

Definition

```
struct drm_dp_vcpi {
  int vcpi;
  int pbn;
  int aligned_pbn;
  int num_slots;
};
```

Members

vcpi Virtual channel ID.

pbn Payload Bandwidth Number for this channel

aligned_pbn PBN aligned with slot size

num_slots number of slots for this PBN

struct **drm_dp_mst_port**
MST port

Definition

```
struct drm_dp_mst_port {
  struct kref kref;
  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 available_pbn;
  struct list_head next;
  struct drm_dp_mst_branch * mstb;
  struct drm_dp_aux aux;
  struct drm_dp_mst_branch * parent;
  struct drm_dp_vcpi vcpi;
  struct drm_connector * connector;
  struct drm_dp_mst_topology_mgr * mgr;
  struct edid * cached_edid;
  bool has_audio;
};
```

Members

kref reference count for this port.

port_num port number

input if this port is an input port.

mcs message capability status - DP 1.2 spec.

ddps DisplayPort Device Plug Status - DP 1.2

pdt Peer Device Type

ldps Legacy Device Plug Status

dpcd_rev DPCD revision of device on this port

num_sdp_streams Number of simultaneous streams

num_sdp_stream_sinks Number of stream sinks

available_pbn Available bandwidth for this port.

next link to next port on this branch device

mstb branch device attach below this port

aux i2c aux transport to talk to device connected to this port.

parent branch device parent of this port

vcpi Virtual Channel Payload info for this port.

connector DRM connector this port is connected to.

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.

has_audio Tracks whether the sink connector to this port is audio-capable.

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 kref;
  u8 rad;
  u8 lct;
  int num_ports;
  int msg_slots;
  struct list_head ports;
  struct drm_dp_mst_port * port_parent;
  struct drm_dp_mst_topology_mgr * mgr;
  struct drm_dp_sideband_msg_tx * tx_slots;
  int last_seqno;
  bool link_address_sent;
  u8 guid;
};
```

Members

kref reference count for this port.

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.

msg_slots one bit per transmitted msg slot.

ports linked list of ports on this branch.

port_parent pointer to the port parent, NULL if toplevel.

mgr topology manager for this branch device.

tx_slots transmission slots for this device.

last_seqno last sequence number used to talk to this.

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_topology_mgr**
DisplayPort MST manager

Definition

```
struct drm_dp_mst_topology_mgr {
  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 down_rep_recv;
  struct drm_dp_sideband_msg_rx up_req_recv;
  struct mutex lock;
  bool mst_state;
  struct drm_dp_mst_branch * mst_primary;
  u8 dpcd;
  u8 sink_count;
  int pbn_div;
  struct drm_dp_mst_topology_state * state;
  const struct drm_private_state_funcs * funcs;
  struct mutex qlock;
  struct list_head tx_msg_downq;
  struct mutex payload_lock;
  struct drm_dp_vcpi ** proposed_vcpis;
  struct drm_dp_payload * payloads;
  unsigned long payload_mask;
  unsigned long vcpi_mask;
  wait_queue_head_t tx_waitq;
  struct work_struct work;
  struct work_struct tx_work;
  struct list_head destroy_connector_list;
  struct mutex destroy_connector_lock;
  struct work_struct destroy_connector_work;
};
```

Members

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 topolgy 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.

down_rep_recv Message receiver state for down replies. This and **up_req_recv** are only ever access from the work item, which is serialised.

up_req_recv Message receiver state for up requests. This and **down_rep_recv** are only ever access from the work item, which is serialised.

lock protects mst state, primary, dpcd.

mst_state If this manager is enabled for an MST capable port. False if no MST sink/branch devices is connected.

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.

pbn_div PBN to slots divisor.

state State information for topology manager

funcs Atomic helper callbacks

qlock protects **tx_msg_downq**, the *drm_dp_mst_branch.txslost* and drm_dp_sideband_msg_tx.state once they are queued

tx_msg_downq List of pending down replies.

payload_lock Protect payload information.

proposed_vcpis Array of pointers for the new VCPI allocation. The VCPI structure itself is *drm_dp_mst_port.vcpi*.

payloads Array of payloads.

payload_mask Elements of **payloads** actually in use. Since reallocation of active outputs isn't possible gaps can be created by disabling outputs out of order compared to how they've been enabled.

vcpi_mask Similar to **payload_mask**, but for **proposed_vcpis**.

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_connector_list List of to be destroyed connectors.

destroy_connector_lock Protects **connector_list**.

destroy_connector_work Work item to destroy connectors. Needed to avoid locking inversion.

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.

int **drm_dp_update_payload_part1**(struct *drm_dp_mst_topology_mgr* * *mgr*)
Execute payload update part 1

Parameters

struct drm_dp_mst_topology_mgr * mgr manager to use.

Description

This iterates over all proposed virtual channels, and tries to allocate space in the link for them. For 0->slots transitions, this step just writes the VCPI to the MST device. For slots->0 transitions, this writes the updated VCPIs and removes the remote VC payloads.

after calling this the driver should generate ACT and payload packets.

int **drm_dp_update_payload_part2**(struct *drm_dp_mst_topology_mgr* * *mgr*)
Execute payload update part 2

Parameters

struct drm_dp_mst_topology_mgr * mgr manager to use.

Description

This iterates over all proposed virtual channels, and tries to allocate space in the link for them. For 0->slots transitions, this step writes the remote VC payload commands. For slots->0 this just resets some internal state.

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*)
resume the MST manager

Parameters

struct drm_dp_mst_topology_mgr * mgr manager to resume

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.

int **drm_dp_mst_hpd_irq**(struct *drm_dp_mst_topology_mgr* * *mgr*, u8 * *esi*, bool * *handled*)
MST hotplug IRQ notify

Parameters

struct drm_dp_mst_topology_mgr * mgr manager to notify irq for.

u8 * esi 4 bytes 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 short IRQ, along with the value of the DEVICE_SERVICE_IRQ_VECTOR_ESI0. The topology manager will process the sideband messages received as a result of this.

enum *drm_connector_status* **drm_dp_mst_detect_port**(struct *drm_connector* * *connector*, 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_dp_mst_topology_mgr * mgr manager for this port

struct drm_dp_mst_port * port unverified pointer to a port

Description

This returns the current connection state for a port. It validates the port pointer still exists so the caller doesn't require a reference

bool **drm_dp_mst_port_has_audio**(struct *drm_dp_mst_topology_mgr* * *mgr*, struct *drm_dp_mst_port* * *port*)
Check whether port has audio capability or not

Parameters

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 whether the port supports audio or not.

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 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_find_vcpi_slots**(struct *drm_dp_mst_topology_mgr* * *mgr*, int *pbn*)
find slots for this PBN value

Parameters

struct drm_dp_mst_topology_mgr * mgr manager to use

int pbn payload bandwidth to convert into slots.

int **drm_dp_atomic_find_vcpi_slots**(struct *drm_atomic_state* * *state*, struct *drm_dp_mst_topology_mgr* * *mgr*, struct *drm_dp_mst_port* * *port*, int *pbn*)
Find and add vcpi 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 vcpi slots for

int pbn bandwidth required for the mode in PBN

Return

Total slots in the atomic state assigned for this port or error

int **drm_dp_atomic_release_vcpi_slots**(struct *drm_atomic_state* * *state*, struct *drm_dp_mst_topology_mgr* * *mgr*, int *slots*)
Release allocated vcpi slots

Parameters

struct drm_atomic_state * state global atomic state

struct drm_dp_mst_topology_mgr * mgr MST topology manager for the port

int slots number of vcpi slots to release

Return

0 if **slots** were added back to `drm_dp_mst_topology_state->avail_slots` or negative error code

bool **drm_dp_mst_allocate_vcpi**(struct *drm_dp_mst_topology_mgr* * *mgr*, struct *drm_dp_mst_port* * *port*, int *pbn*, int *slots*)
Allocate a virtual channel

Parameters

struct drm_dp_mst_topology_mgr * mgr manager for this port

struct drm_dp_mst_port * port port to allocate a virtual channel for.

int pbn payload bandwidth number to request

int slots returned number of slots for this PBN.

void **drm_dp_mst_reset_vcpi_slots**(struct *drm_dp_mst_topology_mgr* * *mgr*, struct *drm_dp_mst_port* * *port*)
Reset number of slots to 0 for VCPI

Parameters

struct drm_dp_mst_topology_mgr * mgr manager for this port

struct drm_dp_mst_port * port unverified pointer to a port.

Description

This just resets the number of slots for the ports VCPI for later programming.

void **drm_dp_mst_deallocate_vcpi**(struct *drm_dp_mst_topology_mgr* * *mgr*, struct *drm_dp_mst_port* * *port*)
deallocate a VCPI

Parameters

struct drm_dp_mst_topology_mgr * mgr manager for this port

struct drm_dp_mst_port * port unverified port to deallocate vcpi for

int **drm_dp_check_act_status**(struct *drm_dp_mst_topology_mgr* * *mgr*)
Check ACT handled status.

Parameters

struct drm_dp_mst_topology_mgr * mgr manager to use

Description

Check the payload status bits in the DPCD for ACT handled completion.

int **drm_dp_calc_pbn_mode**(int *clock*, int *bpp*)
Calculate the PBN for a mode.

Parameters

int clock dot clock for the mode

int bpp bpp for the mode.

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*)

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.

struct drm_dp_mst_topology_state * **drm_atomic_get_mst_topology_state**(struct *drm_atomic_state* * *state*, struct *drm_dp_mst_topology_mgr* * *mgr*)

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. Also, *drm_atomic_get_private_obj_state()* expects the caller to care of the locking, so warn if don't hold the connection_mutex.

Return

The MST topology state or error pointer.

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

5.12 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;
  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 .:c:func:*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 .:c:func:*transfer()* function will seldomly return anything other than the number of bytes contained in the transmit buffer on success.

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;
  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;
  char name;
  unsigned int channel;
  unsigned int lanes;
  enum mipi_dsi_pixel_format format;
  unsigned long mode_flags;
};
```

Members

host DSI host for this peripheral

dev driver model device node for this peripheral

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

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

struct **mipi_dsi_driver**
DSI driver

Definition

```
struct mipi_dsi_driver {
  struct device_driver driver;
  int(* probe) (struct mipi_dsi_device *dsi);
  int(* 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_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

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_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_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.

5.13 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 an error to call this when the output polling support has not yet been set up.

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.

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.
   - debugfs 'override_edid' (used for testing only)
   - firmware EDID (`drm_load_edid_firmware()`)
   - *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_mode_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* helper 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.

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()*.

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 intializes 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_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 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.

5.14 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.

int **drm_edid_header_is_valid**(const u8 * *raw_edid*)
sanity check the header of the base EDID block

Parameters

const u8 * raw_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_block_valid**(u8 * *raw_edid*, int *block*, bool *print_bad_edid*, bool * *edid_corrupt*)
Sanity check the EDID block (base or extension)

Parameters

u8 * raw_edid pointer to raw EDID block

int block 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.

struct edid * **drm_do_get_edid**(struct *drm_connector* * *connector*, int (*get_edid_block) (void **data*, u8 **buf*, unsigned int *block*, size_t *len*, void * *data*)
get EDID data using a custom EDID block read function

Parameters

struct drm_connector * connector connector we're probing

int (*)(void *data,u8 *buf,unsigned int block,size_t len) get_edid_block EDID block read function

void * data 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.

Return

Pointer to valid EDID or NULL if we couldn't find any.

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.

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.

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.

enum hdmi_picture_aspect **drm_get_cea_aspect_ratio**(const u8 *video_code*)
get the picture aspect ratio corresponding to the input VIC from the CEA mode list

Parameters

const u8 video_code ID given to each of the CEA modes

Description

Returns picture aspect ratio

void **drm_edid_get_monitor_name**(struct edid * *edid*, char * *name*, int *bufsize*)
fetch the monitor name from the edid

Parameters

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.)

void **drm_edid_to_eld**(struct *drm_connector* * *connector*, struct edid * *edid*)
build ELD from EDID

Parameters

struct drm_connector * connector connector corresponding to the HDMI/DP sink

struct edid * edid EDID to parse

Description

Fill the ELD (EDID-Like Data) buffer for passing to the audio driver. The Conn_Type, HDCP and Port_ID ELD fields are left for the graphics driver to fill in.

int **drm_edid_to_sad**(struct edid * *edid*, struct cea_sad ** *sads*)
extracts SADs from EDID

Parameters

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**(struct edid * *edid*, u8 ** *sadb*)
extracts Speaker Allocation Data Blocks from EDID

Parameters

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**(struct edid * *edid*)
detect whether monitor is HDMI

Parameters

struct edid * edid monitor EDID information

Description

Parse the CEA extension according to CEA-861-B.

Return

True if the monitor is HDMI, false if not or unknown.

bool **drm_detect_monitor_audio**(struct edid * *edid*)
check monitor audio capability

Parameters

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.

bool **drm_rgb_quant_range_selectable**(struct edid * *edid*)
is RGB quantization range selectable?

Parameters

struct edid * edid EDID block to scan

Description

Check whether the monitor reports the RGB quantization range selection as supported. The AVI infoframe can then be used to inform the monitor which quantization range (full or limited) is used.

Return

True if the RGB quantization range is selectable, 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_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 in **connector** with any information which can be derived from the edid.

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_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_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_display_mode* * *mode*, enum hdmi_quantization_range *rgb_quant_range*, bool *rgb_quant_range_selectable*)
fill the HDMI AVI infoframe quantization range information

Parameters

struct hdmi_avi_infoframe * frame HDMI AVI infoframe

const struct drm_display_mode * mode DRM display mode

enum hdmi_quantization_range rgb_quant_range RGB quantization range (Q)

bool rgb_quant_range_selectable Sink support selectable RGB quantization range (QS)

int **drm_hdmi_vendor_infoframe_from_display_mode**(struct hdmi_vendor_infoframe * *frame*, 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_display_mode * mode DRM display mode

Description

Note that there's is 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.

5.15 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.

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.

bool **drm_scdc_set_scrambling**(struct i2c_adapter * *adapter*, bool *enable*)
enable scrambling

Parameters

struct i2c_adapter * adapter I2C adapter for DDC channel

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 i2c_adapter * *adapter*, bool *set*)
set TMDS clock ratio

Parameters

struct i2c_adapter * adapter I2C adapter for DDC channel

bool set ret or reset the high clock ratio

Description

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.

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 i2c_adapter * *adapter*)
what is status of scrambling?

Parameters

struct i2c_adapter * adapter I2C adapter for DDC channel

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 i2c_adapter * *adapter*, bool *enable*)
enable scrambling

Parameters

struct i2c_adapter * adapter I2C adapter for DDC channel

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 i2c_adapter * *adapter*, bool *set*)
set TMDS clock ratio

Parameters

struct i2c_adapter * adapter I2C adapter for DDC channel

bool set ret or reset the high clock ratio

Description

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.

5.16 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;
  int y1;
  int x2;
  int y2;
};
```

Members

x1 horizontal starting coordinate (inclusive)

y1 vertical starting coordinate (inclusive)

x2 horizontal ending coordinate (exclusive)

y2 vertical ending coordinate (exclusive)

DRM_RECT_FMT()
printf string for *struct drm_rect*

Parameters

DRM_RECT_ARG(*r*)
printf arguments for *struct drm_rect*

Parameters

r rectangle struct

DRM_RECT_FP_FMT()
printf string for *struct drm_rect* in 16.16 fixed point

Parameters

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_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 tranlated

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_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 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.

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*, int *hscale*, int *vscale*)
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

int hscale horizontal scaling factor

int vscale vertical scaling factor

Description

Clip rectangle **dst** by rectangle **clip**. Clip rectangle **src** by the same amounts multiplied by **hscale** and **vscale**.

Return

`true` if rectangle **dst** is still visible after being clipped, `false` otherwise

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).

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).

Return

The vertical scaling factor, or errno of out of limits.

int **drm_rect_calc_hscale_relaxed**(struct *drm_rect* * *src*, struct *drm_rect* * *dst*, int *min_hscale*, int *max_hscale*)
calculate the horizontal scaling factor

Parameters

struct drm_rect * src source window rectangle

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 calculated scaling factor is below **min_vscale**, decrease the height of rectangle **dst** to compensate.

If the calculated scaling factor is above **max_vscale**, decrease the height of rectangle **src** to compensate.

Return

The horizontal scaling factor.

int **drm_rect_calc_vscale_relaxed**(struct *drm_rect* * *src*, struct *drm_rect* * *dst*, int *min_vscale*, int *max_vscale*)
calculate the vertical scaling factor

Parameters

struct drm_rect * src source window rectangle

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 calculated scaling factor is below **min_vscale**, decrease the height of rectangle **dst** to compensate.

If the calculated scaling factor is above **max_vscale**, decrease the height of rectangle **src** to compensate.

Return

The vertical scaling factor.

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 correcsponds 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 correcsponds 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 rotatation and its inverse. That is, if you do

```
DRM_MODE_PROP_ROTATE(:c:type:`r`, width, height, rotation);
DRM_MODE_ROTATE_inv(:c:type:`r`, width, height, rotation);
```

you will always get back the original rectangle.

5.17 HDMI Infoframes Helper Reference

Strictly speaking this is not a DRM helper library but generally useable 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.

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;
};
```

Members

any generic infoframe

avi avi infoframe

spd spd infoframe

vendor union of all vendor infoframes

audio audio 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.

int **hdmi_avi_infoframe_init**(struct hdmi_avi_infoframe * *frame*)
initialize an HDMI AVI infoframe

Parameters

struct hdmi_avi_infoframe * frame HDMI AVI infoframe

Description

Returns 0 on success or a negative error code on failure.

ssize_t **hdmi_avi_infoframe_pack**(struct hdmi_avi_infoframe * *frame*, void * *buffer*, size_t *size*)
write HDMI AVI infoframe to binary buffer

Parameters

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.

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.

ssize_t **hdmi_spd_infoframe_pack**(struct hdmi_spd_infoframe * *frame*, void * *buffer*, size_t *size*)
write HDMI SPD infoframe to binary buffer

Parameters

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.

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.

ssize_t **hdmi_audio_infoframe_pack**(struct hdmi_audio_infoframe * *frame*, void * *buffer*, size_t *size*)
write HDMI audio infoframe to binary buffer

Parameters

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.

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.

ssize_t **hdmi_vendor_infoframe_pack**(struct hdmi_vendor_infoframe * *frame*, void * *buffer*, size_t *size*)
write a HDMI vendor infoframe to binary buffer

Parameters

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_infoframe_pack**(union *hdmi_infoframe* * *frame*, void * *buffer*, size_t *size*)
write a HDMI infoframe to binary buffer

Parameters

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.

void **hdmi_infoframe_log**(const char * *level*, struct device * *dev*, union *hdmi_infoframe* * *frame*)
log info of HDMI infoframe

Parameters

const char * level logging level

struct device * dev device

union hdmi_infoframe * frame HDMI infoframe

int **hdmi_infoframe_unpack**(union *hdmi_infoframe* * *frame*, void * *buffer*)
unpack binary buffer to a HDMI infoframe

Parameters

union hdmi_infoframe * frame HDMI infoframe

void * buffer source 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.

5.18 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 commited;
  spinlock_t lock;
};
```

Members

name debug name

func callback fxn called for each committed item

worker worker which calls **func**

queued queued tasks

commited commited tasks

lock lock to access queued and commited 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

5.19 Plane Helper Reference

This helper library has two parts. The first part has support 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. To avoid too much duplicated code use `drm_plane_helper_check_update()` which can be used to enforce the same restrictions as primary planes had thus. The default primary plane only expose XRBG8888 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 second part also implements transitional helpers which allow drivers to gradually switch to the atomic helper infrastructure for plane updates. Once that switch is complete drivers shouldn't use these any longer, instead using the proper legacy implementations for update and disable plane hooks provided by the atomic helpers.

Again drivers are strongly urged to switch to the new interfaces.

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_check_state**(struct *drm_plane_state* * *state*, const struct *drm_rect* * *clip*, int *min_scale*, int *max_scale*, bool *can_position*, bool *can_update_disabled*)

Check plane state for validity

Parameters

struct drm_plane_state * state plane state to check

const struct drm_rect * clip integer clipping coordinates

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_plane_helper_check_update**(struct *drm_plane* * *plane*, struct *drm_crtc* * *crtc*, struct *drm_framebuffer* * *fb*, struct *drm_rect* * *src*, struct *drm_rect* * *dst*, const struct *drm_rect* * *clip*, unsigned int *rotation*, int *min_scale*, int *max_scale*, bool *can_position*, bool *can_update_disabled*, bool * *visible*)

Check plane update for validity

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

struct drm_rect * src source coordinates in 16.16 fixed point

struct drm_rect * dst integer destination coordinates

const struct drm_rect * clip integer clipping coordinates

unsigned int rotation plane rotation

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?

bool * visible output parameter indicating whether plane is still visible after clipping

Description

Checks that a desired plane update is valid. 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_primary_helper_update**(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

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, not used here

Description

Provides a default plane update handler for primary planes. This is handler is called in response to a userspace SetPlane operation on the plane with a non-NULL framebuffer. We call the driver's modeset handler to update the framebuffer.

`SetPlane()` on a primary plane of a disabled CRTC is not supported, and will return an error.

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.

Drivers for hardware that don't have these restrictions can provide their own implementation rather than using this helper.

Return

Zero on success, error code on failure

int **drm_primary_helper_disable**(struct *drm_plane* * *plane*, struct *drm_modeset_acquire_ctx* * *ctx*)
Helper for primary plane disable

Parameters

struct drm_plane * plane plane to disable

struct drm_modeset_acquire_ctx * ctx lock acquire context, not used here

Description

Provides a default plane disable handler for primary planes. This is handler is called in response to a userspace SetPlane operation on the plane with a NULL framebuffer parameter. It unconditionally fails the disable call with -EINVAL the only way to disable the primary plane without driver support is to disable the entire CRTC. Which does not match the plane `drm_plane_funcs.disable_plane` hook.

Note that some hardware may be able to disable the primary plane without disabling the whole CRTC. Drivers for such hardware should provide their own disable handler that disables just the primary plane (and they'll likely need to provide their own update handler as well to properly re-enable a disabled primary plane).

Return

Unconditionally returns -EINVAL.

void **drm_primary_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_update**(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*)
Transitional helper for plane update

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**

Description

Provides a default plane update handler using the atomic plane update functions. It is fully left to the driver to check plane constraints and handle corner-cases like a fully occluded or otherwise invisible plane.

This is useful for piecewise transitioning of a driver to the atomic helpers.

Return

Zero on success, error code on failure

int **drm_plane_helper_disable**(struct *drm_plane* * *plane*)
Transitional helper for plane disable

Parameters

struct drm_plane * plane plane to disable

Description

Provides a default plane disable handler using the atomic plane update functions. It is fully left to the driver to check plane constraints and handle corner-cases like a fully occluded or otherwise invisible plane.

This is useful for piecewise transitioning of a driver to the atomic helpers.

Return

Zero on success, error code on failure

5.20 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.

Return

Zero on success, error code on failure.

CHAPTER

SIX

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 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 copypasta 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/openeing the primary device node. See also *drm_is_current_master()*.

Clients can authenticate against the current master (if it matches their own) using the GETMAGIC 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 **priv** 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**.

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;
  struct drm_lock_data lock;
  void * driver_priv;
};
```

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`.

lock DRI1 lock information.

driver_priv Pointer to driver-private information.

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 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 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 **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.

6.4 Render nodes

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

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

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

With render nodes, user-space can now control access to the render node via basic file-system access-modes. A running graphics server which authenticates clients on the privileged primary/legacy node is no longer required. Instead, a client can open the render node and is immediately granted GPU access.

Communication between clients (or servers) is done via PRIME. FLINK from render node to legacy node is not supported. New clients must not use the insecure FLINK interface.

Besides dropping all modeset/global ioctls, render nodes also drop the DRM-Master concept. There is no reason to associate render clients with a DRM-Master as they are independent of any graphics server. Besides, they must work without any running master, anyway. Drivers must be able to run without a master object if they support render nodes. If, on the other hand, a driver requires shared state between clients which is visible to user-space and accessible beyond open-file boundaries, they cannot support render nodes.

6.5 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/ioctl/botching-up-ioctls.txt`. 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_IO()`, `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, 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.

typedef int **`drm_ioctl_t`**(struct drm_device * *dev*, void * *data*, struct *drm_file* * *file_priv*)
DRM ioctl function type.

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 alrady copied **data** into kernel-space, and will also copy it back, depending upon the read/write settings in the ioctl command code.

typedef int **drm_ioctl_compat_t**(struct file * *filp*, unsigned int *cmd*, unsigned long *arg*)
compatibility DRM ioctl function type.

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 wreak 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_CONTROL_ALLOW Deprecated, do not use. Control nodes are in the process of getting removed.

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.

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(*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 implemntation

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 implemntation

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.

int **drm_ioctl_permit**(u32 *flags*, struct *drm_file* * *file_priv*)
Check ioctl permissions against caller

Parameters

u32 flags ioctl permission flags.

struct drm_file * file_priv Pointer to struct drm_file identifying the caller.

Description

Checks whether the caller is allowed to run an ioctl with the indicated permissions.

Return

Zero if allowed, -EACCES otherwise.

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.

6.6 Testing and validation

6.6.1 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 can be found in https://cgit.freedesktop.org/drm/igt-gpu-tools/.

To build IGT, start by installing its build dependencies. In Debian-based systems:

```
# apt-get build-dep intel-gpu-tools
```

And in Fedora-based systems:

```
# dnf builddep intel-gpu-tools
```

Then clone the repository:

```
$ git clone git://anongit.freedesktop.org/drm/igt-gpu-tools
```

Configure the build system and start the build:

```
$ cd igt-gpu-tools && ./autogen.sh && make -j6
```

Download the piglit dependency:

```
$ ./scripts/run-tests.sh -d
```

And run the tests:

```
$ ./scripts/run-tests.sh -t kms -t core -s
```

run-tests.sh is a wrapper around piglit that will execute the tests matching the -t options. A report in HTML format will be available in ./results/html/index.html. Results can be compared with piglit.

6.6.2 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*. The debugfs files are automatically set up if that vfunc is set. CRC samples need to be captured in the driver by calling *drm_crtc_add_crc_entry()*.

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.

6.6.3 Debugfs Support

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.

int **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().

6.7 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.

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.

6.8 VBlank event handling

The DRM core exposes two vertical blank related ioctls:

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

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

CHAPTER

SEVEN

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.

7.1 Core Driver Infrastructure

This section covers core driver infrastructure used by both the display and the GEM parts of the driver.

7.1.1 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.

bool **__intel_display_power_is_enabled**(struct drm_i915_private **dev_priv*, enum intel_display_power_domain *domain*)

unlocked check for a power domain

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum intel_display_power_domain domain power domain to check

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.

Return

True when the power domain is enabled, false otherwise.

bool **intel_display_power_is_enabled**(struct drm_i915_private **dev_priv*, enum intel_display_power_domain *domain*)

check for a power domain

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum intel_display_power_domain domain power domain to check

Description

This function can be used to check the hw power domain state. It is mostly used in hardware state readout functions. Everywhere else code should rely upon explicit power domain reference counting to ensure that the hardware block is powered up before accessing it.

Callers must hold the relevant modesetting locks to ensure that concurrent threads can't disable the power well while the caller tries to read a few registers.

Return

True when the power domain is enabled, false otherwise.

void **intel_display_set_init_power**(struct drm_i915_private * *dev_priv*, bool *enable*)
set the initial power domain state

Parameters

struct drm_i915_private * dev_priv i915 device instance

bool enable whether to enable or disable the initial power domain state

Description

For simplicity our driver load/unload and system suspend/resume code assumes that all power domains are always enabled. This functions controls the state of this little hack. While the initial power domain state is enabled runtime pm is effectively disabled.

void **intel_display_power_get**(struct drm_i915_private * *dev_priv*, enum intel_display_power_domain *domain*)
grab a power domain reference

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum intel_display_power_domain domain power domain to reference

Description

This function grabs a power domain reference for **domain** and ensures that the power domain and all its parents are powered up. Therefore users should only grab a reference to the innermost power domain they need.

Any power domain reference obtained by this function must have a symmetric call to *intel_display_power_put()* to release the reference again.

bool **intel_display_power_get_if_enabled**(struct drm_i915_private * *dev_priv*, enum intel_display_power_domain *domain*)
grab a reference for an enabled display power domain

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum intel_display_power_domain domain power domain to reference

Description

This function grabs a power domain reference for **domain** and ensures that the power domain and all its parents are powered up. Therefore users should only grab a reference to the innermost power domain they need.

Any power domain reference obtained by this function must have a symmetric call to *intel_display_power_put()* to release the reference again.

void **intel_display_power_put**(struct drm_i915_private * *dev_priv*, enum intel_display_power_domain *domain*)
release a power domain reference

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum intel_display_power_domain domain power domain to reference

Description

This function drops the power domain reference obtained by *intel_display_power_get()* and might power down the corresponding hardware block right away if this is the last reference.

int **intel_power_domains_init**(struct drm_i915_private * *dev_priv*)
initializes the power domain structures

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

Initializes the power domain structures for **dev_priv** depending upon the supported platform.

void **intel_power_domains_fini**(struct drm_i915_private * *dev_priv*)
finalizes the power domain structures

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

Finalizes the power domain structures for **dev_priv** depending upon the supported platform. This function also disables runtime pm and ensures that the device stays powered up so that the driver can be reloaded.

void **intel_power_domains_init_hw**(struct drm_i915_private * *dev_priv*, bool *resume*)
initialize hardware power domain state

Parameters

struct drm_i915_private * dev_priv i915 device instance

bool resume Called from resume code paths or not

Description

This function initializes the hardware power domain state and enables all power wells belonging to the INIT power domain. Power wells in other domains (and not in the INIT domain) are referenced or disabled during the modeset state HW readout. After that the reference count of each power well must match its HW enabled state, see *intel_power_domains_verify_state()*.

void **intel_power_domains_suspend**(struct drm_i915_private * *dev_priv*)
suspend power domain state

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

This function prepares the hardware power domain state before entering system suspend. It must be paired with *intel_power_domains_init_hw()*.

void **intel_power_domains_verify_state**(struct drm_i915_private * *dev_priv*)
verify the HW/SW state for all power wells

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

Verify if the reference count of each power well matches its HW enabled state and the total refcount of the domains it belongs to. This must be called after modeset HW state sanitization, which is responsible for acquiring reference counts for any power wells in use and disabling the ones left on by BIOS but not required by any active output.

void **intel_runtime_pm_get**(struct drm_i915_private * *dev_priv*)
grab a runtime pm reference

Parameters

struct drm_i915_private * dev_priv i915 device instance

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.

bool **intel_runtime_pm_get_if_in_use**(struct drm_i915_private * *dev_priv*)
grab a runtime pm reference if device in use

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

This function grabs a device-level runtime pm reference if the device is already in use 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.

void **intel_runtime_pm_get_noresume**(struct drm_i915_private * *dev_priv*)
grab a runtime pm reference

Parameters

struct drm_i915_private * dev_priv i915 device instance

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.

void **intel_runtime_pm_put**(struct drm_i915_private * *dev_priv*)
release a runtime pm reference

Parameters

struct drm_i915_private * dev_priv i915 device instance

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 drm_i915_private * *dev_priv*)
enable runtime pm

Parameters

struct drm_i915_private * dev_priv i915 device instance

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 only done on the first modeset using *intel_display_set_init_power()*.

void **intel_uncore_forcewake_get**(struct drm_i915_private * *dev_priv*, enum forcewake_domains *fw_domains*)
grab forcewake domain references

Parameters

struct drm_i915_private * dev_priv i915 device instance

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_unforce_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_get__locked**(struct drm_i915_private * *dev_priv*, enum forcewake_domains *fw_domains*)
grab forcewake domain references

Parameters

struct drm_i915_private * dev_priv i915 device instance

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 drm_i915_private * *dev_priv*, enum forcewake_domains *fw_domains*)
release a forcewake domain reference

Parameters

struct drm_i915_private * dev_priv i915 device instance

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_put__locked**(struct drm_i915_private * *dev_priv*, enum forcewake_domains *fw_domains*)
grab forcewake domain references

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum forcewake_domains fw_domains forcewake domains to get reference on

Description

See *intel_uncore_forcewake_put()*. This variant places the onus on the caller to explicitly handle the dev_priv->uncore.lock spinlock.

int **gen6_reset_engines**(struct drm_i915_private * *dev_priv*, unsigned *engine_mask*)
reset individual engines

Parameters

struct drm_i915_private * dev_priv i915 device

unsigned engine_mask mask of `intel_ring_flag()` engines or ALL_ENGINES for full reset

Description

This function will reset the individual engines that are set in engine_mask. If you provide ALL_ENGINES as mask, full global domain reset will be issued.

Note

It is responsibility of the caller to handle the difference between asking full domain reset versus reset for all available individual engines.

Returns 0 on success, nonzero on error.

int **__intel_wait_for_register_fw**(struct drm_i915_private * *dev_priv*, 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 drm_i915_private * dev_priv the i915 device

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

```
(I915_READ_FW(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 20,0000 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).

Returns 0 if the register matches the desired condition, or -ETIMEOUT.

int **intel_wait_for_register**(struct drm_i915_private * *dev_priv*, i915_reg_t *reg*, u32 *mask*, u32 *value*, unsigned int *timeout_ms*)
wait until register matches expected state

Parameters

struct drm_i915_private * dev_priv the i915 device

i915_reg_t reg the register to read

u32 mask mask to apply to register value

u32 value expected value

unsigned int timeout_ms 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

```
(I915_READ(reg) & mask) == value
```

Otherwise, the wait will timeout after **timeout_ms** milliseconds.

Returns 0 if the register matches the desired condition, or -ETIMEOUT.

enum forcewake_domains **intel_uncore_forcewake_for_reg**(struct drm_i915_private * *dev_priv*, i915_reg_t *reg*, unsigned int *op*)
which forcewake domains are needed to access a register

Parameters

struct drm_i915_private * dev_priv pointer to struct drm_i915_private

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.

7.1.2 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.

7.1.3 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 **i915_check_vgpu**(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 drm_i915_private * *dev_priv*)
deballoon reserved graphics address trunks

Parameters

struct drm_i915_private * dev_priv i915 device private data

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 drm_i915_private * *dev_priv*)
balloon out reserved graphics address trunks

Parameters

struct drm_i915_private * dev_priv i915 device private data

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.

```
                         vGPU1 view         Host view
            0 ------> +-----------+     +-----------+
              ^       |###########|     |   vGPU3   |
              |       |###########|     +-----------+
              |       |###########|     |   vGPU2   |
              |       +-----------+     +-----------+
       mappable GM    | available | ==> |   vGPU1   |
              |       +-----------+     +-----------+
              |       |###########|     |           |
              v       |###########|     |   Host    |
              +=======+===========+     +===========+
              ^       |###########|     |   vGPU3   |
              |       |###########|     +-----------+
              |       |###########|     |   vGPU2   |
              |       +-----------+     +-----------+
     unmappable GM    | available | ==> |   vGPU1   |
              |       +-----------+     +-----------+
              |       |###########|     |           |
              |       |###########|     |   Host    |
              v       |###########|     |           |
total GM size ------> +-----------+     +-----------+
```

Return

zero on success, non-zero if configuration invalid or ballooning failed

7.1.4 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.

void **intel_gvt_sanitize_options**(struct drm_i915_private * *dev_priv*)
sanitize GVT related options

Parameters

struct drm_i915_private * dev_priv drm i915 private data

Description

This function is called at the i915 options sanitize stage.

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_cleanup**(struct drm_i915_private * *dev_priv*)
cleanup GVT components when i915 driver is unloading

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.

7.2 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.

7.2.1 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.

7.2.2 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 GEM tracks frontbuffers using a bitmask for all possible frontbuffer slots through *i915_gem_track_fb()*. The function 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 int.

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.

Note that there's also an older frontbuffer activity tracking scheme which just tracks general activity. This is done by the various mark_busy and mark_idle functions. For display power management features using these functions is deprecated and should be avoided.

bool **intel_fb_obj_invalidate**(struct drm_i915_gem_object * *obj*, enum fb_op_origin *origin*)
invalidate frontbuffer object

Parameters

struct drm_i915_gem_object * obj 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_fb_obj_flush**(struct drm_i915_gem_object * *obj*, enum fb_op_origin *origin*)
flush frontbuffer object

Parameters

struct drm_i915_gem_object * obj 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 **intel_frontbuffer_flush**(struct drm_i915_private * *dev_priv*, unsigned *frontbuffer_bits*, enum fb_op_origin *origin*)
flush frontbuffer

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

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 * *dev_priv*, unsigned *frontbuffer_bits*)
prepare asynchronous frontbuffer flip

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*, unsigned *frontbuffer_bits*)
complete asynchronous frontbuffer flip

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*, unsigned *frontbuffer_bits*)
synchronous frontbuffer flip

Parameters

struct drm_i915_private * dev_priv 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 **i915_gem_track_fb**(struct drm_i915_gem_object * *old*, struct drm_i915_gem_object * *new*, unsigned *frontbuffer_bits*)
update frontbuffer tracking

Parameters

struct drm_i915_gem_object * old current GEM buffer for the frontbuffer slots

struct drm_i915_gem_object * new new GEM buffer for the frontbuffer slots

unsigned 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.

7.2.3 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 underrrun 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 transcoder *pch_transcoder*, bool *enable*)
set PCH fifo underrun reporting state

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum transcoder 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 interruts 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 transcoder *pch_transcoder*)
handle PCH fifo underrun interrupt

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum transcoder 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.

7.2.4 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.

7.2.5 Atomic Plane Helpers

The functions here are used by the atomic plane helper functions to implement legacy plane updates (i.e., drm_plane->:c:func:*update_plane()* and drm_plane->:c:func:*disable_plane()*). This allows plane updates to use the atomic state infrastructure and perform plane updates as separate prepare/check/commit/cleanup steps.

struct intel_plane_state * **intel_create_plane_state**(struct *drm_plane* * *plane*)
create plane state object

Parameters

struct drm_plane * plane drm plane

Description

Allocates a fresh plane state for the given plane and sets some of the state values to sensible initial values.

Return

A newly allocated plane state, or NULL on failure

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_plane_atomic_get_property**(struct *drm_plane* * *plane*, const struct *drm_plane_state* * *state*, struct *drm_property* * *property*, uint64_t * *val*)
fetch plane property value

Parameters

struct drm_plane * plane plane to fetch property for

const struct drm_plane_state * state state containing the property value

struct drm_property * property property to look up

uint64_t * val pointer to write property value into

Description

The DRM core does not store shadow copies of properties for atomic-capable drivers. This entrypoint is used to fetch the current value of a driver-specific plane property.

int **intel_plane_atomic_set_property**(struct *drm_plane* * *plane*, struct *drm_plane_state* * *state*, struct *drm_property* * *property*, uint64_t *val*)

set plane property value

Parameters

struct drm_plane * plane plane to set property for

struct drm_plane_state * state state to update property value in

struct drm_property * property property to set

uint64_t val value to set property to

Description

Writes the specified property value for a plane into the provided atomic state object.

Returns 0 on success, -EINVAL on unrecognized properties

7.2.6 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.

7.2.7 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.

bool **intel_hpd_irq_storm_detect**(struct drm_i915_private * *dev_priv*, enum hpd_pin *pin*)
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

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**->hotplug.hpd_storm_threshold which defaults to **HPD_STORM_DEFAULT_THRESHOLD**. If this threshold is exceeded, it's considered an irq storm and the irq state is set to **HPD_MARK_DISABLED**.

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_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_init()*, which enables connector polling

void **intel_hpd_poll_init**(struct drm_i915_private * *dev_priv*)
enables/disables polling for connectors with hpd

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

This function enables polling for all connectors, regardless of whether or not they support hotplug detection. 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 seperate worker.

Also see: *intel_hpd_init()*, which restores hpd handling.

7.2.8 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 *i915_audio_component* is used to interact between the graphics and audio drivers. The struct *i915_audio_component_ops* **ops** in it is defined in graphics driver and called in audio driver. The struct *i915_audio_component_audio_ops* **audio_ops** is called from i915 driver.

void **intel_audio_codec_enable**(struct intel_encoder * *intel_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 * intel_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 * *intel_encoder*)
Disable the audio codec for HD audio

Parameters

struct intel_encoder * intel_encoder encoder on which to disable audio

Description

The disable sequences must be performed before disabling the transcoder or port.

void **intel_init_audio_hooks**(struct drm_i915_private * *dev_priv*)
Set up chip specific audio hooks

Parameters

struct drm_i915_private * dev_priv device private

void **i915_audio_component_init**(struct drm_i915_private * *dev_priv*)
initialize and register the audio component

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*)
deregister the audio component

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*)
Initialize the audio driver either using component framework or using lpe audio bridge

Parameters

struct drm_i915_private * dev_priv the i915 drm device private data

void **intel_audio_deinit**(struct drm_i915_private * *dev_priv*)
deinitialize the audio driver

Parameters

struct drm_i915_private * dev_priv the i915 drm device private data

struct **i915_audio_component_ops**
Ops implemented by i915 driver, called by hda driver

Definition

```
struct i915_audio_component_ops {
  struct module * owner;
  void (* get_power) (struct device *);
  void (* put_power) (struct device *);
  void (* codec_wake_override) (struct device *, bool enable);
  int (* get_cdclk_freq) (struct device *);
  int (* sync_audio_rate) (struct device *, int port, int pipe, int rate);
  int (* get_eld) (struct device *, int port, int pipe, bool *enabled, unsigned char *buf, int␣
→max_bytes);
};
```

Members

owner i915 module

get_power get the POWER_DOMAIN_AUDIO power well

Request the power well to be turned on.

put_power put the POWER_DOMAIN_AUDIO power well

Allow the power well to be turned off.

codec_wake_override Enable/disable codec wake signal

get_cdclk_freq Get the Core Display Clock in kHz

sync_audio_rate set n/cts based on the sample rate

Called from audio driver. After audio driver sets the sample rate, it will call this function to set n/cts

get_eld fill the audio state and ELD bytes for the given port

Called from audio driver to get the HDMI/DP audio state of the given digital port, and also fetch ELD bytes to the given pointer.

It returns the byte size of the original ELD (not the actually copied size), zero for an invalid ELD, or a negative error code.

Note that the returned size may be over **max_bytes**. Then it implies that only a part of ELD has been copied to the buffer.

struct **i915_audio_component_audio_ops**
Ops implemented by hda driver, called by i915 driver

Definition

```
struct i915_audio_component_audio_ops {
  void * audio_ptr;
  void (* pin_eld_notify) (void *audio_ptr, int port, int pipe);
};
```

Members

audio_ptr Pointer to be used in call to pin_eld_notify

pin_eld_notify Notify the HDA driver that pin sense and/or ELD information has changed

Called when the i915 driver has set up audio pipeline or has just begun to tear it down. This allows the HDA driver to update its status accordingly (even when the HDA controller is in power save mode).

struct **i915_audio_component**
Used for direct communication between i915 and hda drivers

Definition

```
struct i915_audio_component {
  struct device * dev;
  int aud_sample_rate;
  const struct i915_audio_component_ops * ops;
  const struct i915_audio_component_audio_ops * audio_ops;
};
```

Members

dev i915 device, used as parameter for ops

aud_sample_rate the array of audio sample rate per port

ops Ops implemented by i915 driver, called by hda driver

audio_ops Ops implemented by hda driver, called by i915 driver

7.2.9 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_teardown**(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 pipe *pipe*, 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 pipe pipe pipe

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.

7.2.10 Panel Self Refresh PSR (PSR/SRD)

Since Haswell Display controller supports Panel Self-Refresh on display panels witch 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 frontbuffer 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 frontbuffer 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 frontbuffer 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."

void **intel_psr_enable**(struct intel_dp * *intel_dp*)
Enable PSR

Parameters

struct intel_dp * intel_dp Intel DP

Description

This function can only be called after the pipe is fully trained and enabled.

void **intel_psr_disable**(struct intel_dp * *intel_dp*)
Disable PSR

Parameters

struct intel_dp * intel_dp Intel DP

Description

This function needs to be called before disabling pipe.

void **intel_psr_single_frame_update**(struct drm_i915_private * *dev_priv*, unsigned *frontbuffer_bits*)
Single Frame Update

Parameters

struct drm_i915_private * dev_priv i915 device

unsigned frontbuffer_bits frontbuffer plane tracking bits

Description

Some platforms support a single frame update feature that is used to send and update only one frame on Remote Frame Buffer. So far it is only implemented for Valleyview and Cherryview because hardware requires this to be done before a page flip.

void **intel_psr_invalidate**(struct drm_i915_private * *dev_priv*, unsigned *frontbuffer_bits*)
Invalidade PSR

Parameters

struct drm_i915_private * dev_priv i915 device

unsigned frontbuffer_bits frontbuffer plane tracking bits

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 drm_i915_private * *dev_priv*)
Init basic PSR work and mutex.

Parameters

struct drm_i915_private * dev_priv i915 device private

Description

This function is called only once at driver load to initialize basic PSR stuff.

7.2.11 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.

bool **intel_fbc_is_active**(struct drm_i915_private * *dev_priv*)
Is FBC active?

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

This function is used to verify the current state of FBC.

FIXME: This should be tracked in the plane config eventually instead of queried at runtime for most callers.

void **intel_fbc_choose_crtc**(struct drm_i915_private * *dev_priv*, struct *drm_atomic_state* * *state*)
select a CRTC to enable FBC on

Parameters

struct drm_i915_private * dev_priv i915 device instance

struct drm_atomic_state * state the atomic state structure

Description

This function looks at the proposed state for CRTCs and planes, then chooses which pipe is going to have FBC by setting intel_crtc_state->enable_fbc to true.

Later, intel_fbc_enable is going to look for state->enable_fbc and then maybe enable FBC for the chosen CRTC. If it does, it will set dev_priv->fbc.crtc.

void **intel_fbc_enable**(struct intel_crtc * *crtc*, struct intel_crtc_state * *crtc_state*, struct intel_plane_state * *plane_state*)

Parameters

struct intel_crtc * crtc the CRTC

struct intel_crtc_state * crtc_state corresponding *drm_crtc_state* for **crtc**

struct intel_plane_state * plane_state corresponding *drm_plane_state* for the primary plane of **crtc**

Description

This function checks if the given CRTC was chosen for FBC, then enables it if possible. Notice that it doesn't activate FBC. It is valid to call intel_fbc_enable multiple times for the same pipe without an intel_fbc_disable in the middle, as long as it is deactivated.

void **__intel_fbc_disable**(struct drm_i915_private * *dev_priv*)
disable FBC

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

This is the low level function that actually disables FBC. Callers should grab the FBC lock.

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_global_disable**(struct drm_i915_private * *dev_priv*)
globally disable FBC

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

This function disables FBC regardless of which CRTC is associated with it.

void **intel_fbc_handle_fifo_underrun_irq**(struct drm_i915_private * *dev_priv*)
disable FBC when we get a FIFO underrun

Parameters

struct drm_i915_private * dev_priv i915 device instance

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_pipe_state**(struct drm_i915_private * *dev_priv*)
initialize FBC's CRTC visibility tracking

Parameters

struct drm_i915_private * dev_priv i915 device instance

Description

The FBC code needs to track CRTC visibility since the older platforms can't have FBC enabled while multiple pipes are used. This function does the initial setup at driver load to make sure FBC is matching the real hardware.

void **intel_fbc_init**(struct drm_i915_private * *dev_priv*)
Initialize FBC

Parameters

struct drm_i915_private * dev_priv the i915 device

Description

This function might be called during PM init process.

7.2.12 Display Refresh Rate Switching (DRRS)

Display Refresh Rate Switching (DRRS) is a power conservation feature which enables swtching 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_edp_drrs_invalidate()* and *intel_edp_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_dp_set_drrs_state**(struct drm_i915_private * *dev_priv*, struct intel_crtc_state * *crtc_state*, int *refresh_rate*)
program registers for RR switch to take effect

Parameters

struct drm_i915_private * dev_priv i915 device

struct intel_crtc_state * crtc_state a pointer to the active intel_crtc_state

int refresh_rate RR to be programmed

Description

This function gets called when refresh rate (RR) has to be changed from one frequency to another. Switches can be between high and low RR supported by the panel or to any other RR based on media playback (in this case, RR value needs to be passed from user space).

The caller of this function needs to take a lock on dev_priv->drrs.

void **intel_edp_drrs_enable**(struct intel_dp * *intel_dp*, struct intel_crtc_state * *crtc_state*)
init drrs struct if supported

Parameters

struct intel_dp * intel_dp DP struct

struct intel_crtc_state * crtc_state A pointer to the active crtc state.

Description

Initializes frontbuffer_bits and drrs.dp

void **intel_edp_drrs_disable**(struct intel_dp * *intel_dp*, struct intel_crtc_state * *old_crtc_state*)
Disable DRRS

Parameters

struct intel_dp * intel_dp DP struct

struct intel_crtc_state * old_crtc_state Pointer to old crtc_state.

void **intel_edp_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_edp_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.

struct *drm_display_mode* * **intel_dp_drrs_init**(struct intel_connector * *intel_connector*, struct *drm_display_mode* * *fixed_mode*)
Init basic DRRS work and mutex.

Parameters

struct intel_connector * intel_connector eDP connector

struct drm_display_mode * fixed_mode preferred mode of panel

Description

This function is called only once at driver load to initialize basic DRRS stuff.

Return

Downclock mode if panel supports it, else return NULL. DRRS support is determined by the presence of downclock mode (apart from VBT setting).

7.2.13 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 trough 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.

Eeach 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      |
|  CMN/PLL/REF  |  CMN/PLL/REF  |
|---------------|---------------| Display PHY
| PCS01 | PCS23 | PCS01 | PCS23 |
|-------|-------|-------|-------|
|TX0|TX1|TX2|TX3|TX0|TX1|TX2|TX3|
---------------------------------
|     DDI0      |     DDI1      | DP/HDMI ports
---------------------------------

Single channel PHY (CHV/BXT)
-----------------
|      CH0      |
|  CMN/PLL/REF  |
|---------------| Display PHY
| PCS01 | PCS23 |
|-------|-------|
|TX0|TX1|TX2|TX3|
-----------------
|     DDI2      | DP/HDMI port
-----------------
```

7.2.14 CSR firmware support for DMC

Display Context Save and Restore (CSR) firmware support added from gen9 onwards to drive 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_csr_load_program**(struct drm_i915_private * *dev_priv*)
write the firmware from memory to register.

Parameters

struct drm_i915_private * dev_priv i915 drm device.

Description

CSR 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_csr_ucode_init**(struct drm_i915_private * *dev_priv*)
initialize the firmware loading.

Parameters

struct drm_i915_private * dev_priv 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_csr_ucode_suspend**(struct drm_i915_private * *dev_priv*)
prepare CSR firmware before system suspend

Parameters

struct drm_i915_private * dev_priv 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_csr_ucode_resume**(struct drm_i915_private * *dev_priv*)
init CSR firmware during system resume

Parameters

struct drm_i915_private * dev_priv i915 drm device

Description

Reinitialize the DMC firmware during system resume, reacquiring any resources released in *intel_csr_ucode_suspend()*.

void **intel_csr_ucode_fini**(struct drm_i915_private * *dev_priv*)
unload the CSR firmware.

Parameters

struct drm_i915_private * dev_priv i915 drm device.

Description

Firmmware unloading includes freeing the internal memory and reset the firmware loading status.

7.2.15 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 * *dev_priv*)
find VBT and initialize settings from the BIOS

Parameters

struct drm_i915_private * dev_priv 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.

bool **intel_bios_is_tv_present**(struct drm_i915_private * *dev_priv*)
is integrated TV present in VBT

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*, u8 * *i2c_pin*)
is LVDS present in VBT

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*, enum port *port*)
is the specified digital port present

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum port port port to check

Description

Return true if the device in `port` is present.

bool **intel_bios_is_port_edp**(struct drm_i915_private * *dev_priv*, enum port *port*)
is the device in given port eDP

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum port port port to check

Description

Return true if the device in `port` is eDP.

bool **intel_bios_is_dsi_present**(struct drm_i915_private * *dev_priv*, enum port * *port*)
is DSI present in VBT

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum port * port port for DSI if present

Description

Return true if DSI is present, and return the port in `port`.

bool **intel_bios_is_port_hpd_inverted**(struct drm_i915_private * *dev_priv*, enum port *port*)
is HPD inverted for `port`

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum port port port to check

Description

Return true if HPD should be inverted for `port`.

bool **intel_bios_is_lspcon_present**(struct drm_i915_private * *dev_priv*, enum port *port*)
if LSPCON is attached on `port`

Parameters

struct drm_i915_private * dev_priv i915 device instance

enum port port port to check

Description

Return true if LSPCON is present on this port

struct **vbt_header**
VBT Header structure

Definition

```
struct vbt_header {
  u8 signature;
  u16 version;
  u16 header_size;
  u16 vbt_size;
  u8 vbt_checksum;
  u8 reserved0;
  u32 bdb_offset;
  u32 aim_offset;
};
```

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_offset Offset 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;
  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)

7.2.16 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 **skl_init_cdclk**(struct drm_i915_private * *dev_priv*)
Initialize CDCLK on SKL

Parameters

struct drm_i915_private * dev_priv i915 device

Description

Initialize CDCLK for SKL and derivatives. 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 **skl_uninit_cdclk**(struct drm_i915_private * *dev_priv*)
Uninitialize CDCLK on SKL

Parameters

struct drm_i915_private * dev_priv i915 device

Description

Uninitialize CDCLK for SKL and derivatives. This is done only during the display core uninitialization sequence.

void **bxt_init_cdclk**(struct drm_i915_private * *dev_priv*)
Initialize CDCLK on BXT

Parameters

struct drm_i915_private * dev_priv i915 device

Description

Initialize CDCLK for BXT and derivatives. 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 **bxt_uninit_cdclk**(struct drm_i915_private * *dev_priv*)
Uninitialize CDCLK on BXT

Parameters

struct drm_i915_private * dev_priv i915 device

Description

Uninitialize CDCLK for BXT and derivatives. This is done only during the display core uninitialization sequence.

void **cnl_init_cdclk**(struct drm_i915_private * *dev_priv*)
Initialize CDCLK on CNL

Parameters

struct drm_i915_private * dev_priv i915 device

Description

Initialize CDCLK for CNL. 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 **cnl_uninit_cdclk**(struct drm_i915_private * *dev_priv*)
Uninitialize CDCLK on CNL

Parameters

struct drm_i915_private * dev_priv i915 device

Description

Uninitialize CDCLK for CNL. This is done only during the display core uninitialization sequence.

bool **intel_cdclk_state_compare**(const struct intel_cdclk_state * *a*, const struct intel_cdclk_state * *b*)
Determine if two CDCLK states differ

Parameters

const struct intel_cdclk_state * a first CDCLK state

const struct intel_cdclk_state * b second CDCLK state

Return

True if the CDCLK states are identical, false if they differ.

void **intel_set_cdclk**(struct drm_i915_private * *dev_priv*, const struct intel_cdclk_state * *cdclk_state*)
Push the CDCLK state to the hardware

Parameters

struct drm_i915_private * dev_priv i915 device

const struct intel_cdclk_state * cdclk_state new CDCLK state

Description

Program the hardware based on the passed in 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.

void **intel_update_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 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

7.2.17 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 modest interface. During an atomic operation, a PLL can be requested for a given CRTC and encoder configuration by calling *intel_get_shared_dpll()* and a previously used PLL can be released with *intel_release_shared_dpll()*. 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**

enum *intel_dpll_id* **intel_get_shared_dpll_id**(struct drm_i915_private * *dev_priv*, struct *intel_shared_dpll* * *pll*)
get the id of a DPLL

Parameters

struct drm_i915_private * dev_priv i915 device instance

struct intel_shared_dpll * pll the DPLL

Return

The id of **pll**

void **intel_prepare_shared_dpll**(struct intel_crtc * *crtc*)
call a dpll's prepare hook

Parameters

struct intel_crtc * crtc CRTC which has a shared dpll

Description

This calls the PLL's prepare hook if it has one and if the PLL is not already enabled. The prepare hook is platform specific.

void **intel_enable_shared_dpll**(struct intel_crtc * *crtc*)
enable a CRTC's shared DPLL

Parameters

struct intel_crtc * crtc CRTC which has a shared DPLL

Description

Enable the shared DPLL used by **crtc**.

void **intel_disable_shared_dpll**(struct intel_crtc * *crtc*)
disable a CRTC's shared DPLL

Parameters

struct intel_crtc * crtc CRTC which has a shared DPLL

Description

Disable the shared DPLL used by **crtc**.

void **intel_shared_dpll_swap_state**(struct *drm_atomic_state* * *state*)
make atomic DPLL configuration effective

Parameters

struct drm_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 **intel_shared_dpll_init**(struct drm_device * *dev*)
Initialize shared DPLLs

Parameters

struct drm_device * dev drm device

Description

Initialize shared DPLLs for **dev**.

struct *intel_shared_dpll* * **intel_get_shared_dpll**(struct intel_crtc * *crtc*, struct intel_crtc_state * *crtc_state*, struct intel_encoder * *encoder*)
get a shared DPLL for CRTC and encoder combination

Parameters

struct intel_crtc * crtc CRTC

struct intel_crtc_state * crtc_state atomic state for **crtc**

struct intel_encoder * encoder encoder

Description

Find an appropriate DPLL for the given CRTC and encoder combination. A reference from the **crtc** to the returned pll is registered in the atomic state. That configuration is made effective by calling *intel_shared_dpll_swap_state()*. The reference should be released by calling *intel_release_shared_dpll()*.

Return

A shared DPLL to be used by **crtc** and **encoder** with the given **crtc_state**.

void **intel_release_shared_dpll**(struct *intel_shared_dpll* * *dpll*, struct intel_crtc * *crtc*, struct *drm_atomic_state* * *state*)
end use of DPLL by CRTC in atomic state

Parameters

struct intel_shared_dpll * dpll dpll in use by **crtc**

struct intel_crtc * crtc crtc

struct drm_atomic_state * state atomic state

Description

This function releases the reference from **crtc** to **dpll** from the atomic **state**. The new configuration is made effective by calling *intel_shared_dpll_swap_state()*.

void **intel_dpll_dump_hw_state**(struct drm_i915_private * *dev_priv*, struct intel_dpll_hw_state * *hw_state*)
write hw_state to dmesg

Parameters

struct drm_i915_private * dev_priv i915 drm device

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_DEBUG_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

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 {
  unsigned crtc_mask;
  struct intel_dpll_hw_state hw_state;
};
```

Members

crtc_mask mask of CRTC 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_get_shared_dpll()* and *intel_release_shared_dpll()*.

struct **intel_shared_dpll_funcs**
platform specific hooks for managing DPLLs

Definition

```
struct intel_shared_dpll_funcs {
  void (* prepare) (struct drm_i915_private *dev_priv, struct intel_shared_dpll *pll);
  void (* enable) (struct drm_i915_private *dev_priv, struct intel_shared_dpll *pll);
  void (* disable) (struct drm_i915_private *dev_priv, struct intel_shared_dpll *pll);
  bool (* get_hw_state) (struct drm_i915_private *dev_priv,struct intel_shared_dpll *pll,␣
↪struct intel_dpll_hw_state *hw_state);
};
```

Members

prepare Optional hook to perform operations prior to enabling the PLL. Called from *intel_prepare_shared_dpll()* function unless the PLL is already enabled.

enable Hook for enabling the pll, called from *intel_enable_shared_dpll()* if the pll is not already enabled.

disable Hook for disabling the pll, called from *intel_disable_shared_dpll()* only when it is safe to disable the pll, i.e., there are no more tracked users for it.

get_hw_state Hook for reading the values currently programmed to the DPLL registers. This is used for initial hw state readout and state verification after a mode set.

struct **intel_shared_dpll**
display PLL with tracked state and users

Definition

```
struct intel_shared_dpll {
  struct intel_shared_dpll_state state;
  unsigned active_mask;
  bool on;
  const char * name;
  enum intel_dpll_id id;
  struct intel_shared_dpll_funcs funcs;
#define INTEL_DPLL_ALWAYS_ON  (1 \\\lt;\\\lt; 0
  uint32_t flags;
};
```

Members

state Store the state for the pll, including the its hw state and CRTCs using it.

active_mask mask of active CRTCs (i.e. DPMS on) using this DPLL

on is the PLL actually active? Disabled during modeset

name DPLL name; used for logging

id unique indentifier for this DPLL; should match the index in the dev_priv->shared_dplls array

funcs platform specific hooks

flags

INTEL_DPLL_ALWAYS_ON Inform the state checker that the DPLL is kept enabled even if not in use by any CRTC.

7.3 Memory Management and Command Submission

This sections covers all things related to the GEM implementation in the i915 driver.

7.3.1 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 generally rejects such commands, though it may allow some from the drm master process.

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 (for both normal and drm master processes).

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, master-only, register whitelist. The parser implements a number of checks, including the privileged memory checks, via a general bitmasking mechanism.

void **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 drm_i915_gem_object * *batch_obj*, struct drm_i915_gem_object * *shadow_batch_obj*, u32 *batch_start_offset*, u32 *batch_len*, bool *is_master*)
parse a submitted batch buffer for privilege violations

Parameters

struct intel_engine_cs * engine the engine on which the batch is to execute

struct drm_i915_gem_object * batch_obj the batch buffer in question

struct drm_i915_gem_object * shadow_batch_obj copy of the batch buffer in question

u32 batch_start_offset byte offset in the batch at which execution starts

u32 batch_len length of the commands in batch_obj

bool is_master is the submitting process the drm master?

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

7.3.2 Batchbuffer Pools

In order to submit batch buffers as 'secure', the software command parser must ensure that a batch buffer cannot be modified after parsing. It does this by copying the user provided batch buffer contents to a kernel owned buffer from which the hardware will actually execute, and by carefully managing the address space bindings for such buffers.

The batch pool framework provides a mechanism for the driver to manage a set of scratch buffers to use for this purpose. The framework can be extended to support other uses cases should they arise.

void **i915_gem_batch_pool_init**(struct intel_engine_cs * *engine*, struct i915_gem_batch_pool * *pool*)
initialize a batch buffer pool

Parameters

struct intel_engine_cs * engine the associated request submission engine

struct i915_gem_batch_pool * pool the batch buffer pool

void **i915_gem_batch_pool_fini**(struct i915_gem_batch_pool * *pool*)
clean up a batch buffer pool

Parameters

struct i915_gem_batch_pool * pool the pool to clean up

Note

Callers must hold the struct_mutex.

struct drm_i915_gem_object * **i915_gem_batch_pool_get**(struct i915_gem_batch_pool * *pool*, size_t *size*)
allocate a buffer from the pool

Parameters

struct i915_gem_batch_pool * pool the batch buffer pool

size_t size the minimum desired size of the returned buffer

Description

Returns an inactive buffer from **pool** with at least **size** bytes, with the pages pinned. The caller must `i915_gem_object_unpin_pages()` on the returned object.

Note

Callers must hold the struct_mutex

Return

the buffer object or an error pointer

7.3.3 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 PPGTTs (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 requests 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).

int **intel_sanitize_enable_execlists**(struct drm_i915_private * *dev_priv*, int *enable_execlists*)
sanitize i915.enable_execlists

Parameters

struct drm_i915_private * dev_priv i915 device private

int enable_execlists value of i915.enable_execlists module parameter.

Description

Only certain platforms support Execlists (the prerequisites being support for Logical Ring Contexts and Aliasing PPGTT or better).

Return

1 if Execlists is supported and has to be enabled.

void **intel_lr_context_descriptor_update**(struct i915_gem_context * *ctx*, struct intel_engine_cs * *engine*)

calculate & cache the descriptor descriptor for a pinned context

Parameters

struct i915_gem_context * ctx Context to work on

struct intel_engine_cs * engine Engine the descriptor will be used with

Description

The context descriptor encodes various attributes of a context, including its GTT address and some flags. Because it's fairly expensive to calculate, we'll just do it once and cache the result, which remains valid until the context is unpinned.

This is what a descriptor looks like, from LSB to MSB:

```
bits  0-11:    flags, GEN8_CTX_* (cached in ctx->desc_template)
bits 12-31:    LRCA, GTT address of (the HWSP of) this context
bits 32-52:    ctx ID, a globally unique tag
bits 53-54:    mbz, reserved for use by hardware
bits 55-63:    group ID, currently unused and set to 0
```

void **intel_logical_ring_cleanup**(struct intel_engine_cs * *engine*)

deallocate the Engine Command Streamer

Parameters

struct intel_engine_cs * engine Engine Command Streamer.

7.3.4 Global GTT views

Background and previous state

Historically objects could exists (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_ggtt_view_type and struct i915_ggtt_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_ggtt_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_ggtt_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_ggtt_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.ggtt_view and exists for the lifetime of an VMA.

Core API is designed to have copy semantics which means that passed in struct i915_ggtt_view does not need to be persistent (left around after calling the core API functions).

void **i915_ggtt_cleanup_hw**(struct drm_i915_private * *dev_priv*)
Clean up GGTT hardware initialization

Parameters

struct drm_i915_private * dev_priv i915 device

int **i915_ggtt_probe_hw**(struct drm_i915_private * *dev_priv*)
Probe GGTT hardware location

Parameters

struct drm_i915_private * dev_priv i915 device

int **i915_ggtt_init_hw**(struct drm_i915_private * *dev_priv*)
Initialize GGTT hardware

Parameters

struct drm_i915_private * dev_priv i915 device

int **i915_gem_gtt_reserve**(struct i915_address_space * *vm*, 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 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 *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 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 is 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.

7.3.5 GTT Fences and Swizzling

int **i915_vma_put_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.

Return

0 on success, negative error code on failure.

int **i915_vma_get_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.

Return

0 on success, negative error code on failure.

void **i915_gem_revoke_fences**(struct drm_i915_private * *dev_priv*)
revoke fence state

Parameters

struct drm_i915_private * dev_priv i915 device private

Description

Removes all GTT mmappings via the fence registers. This forces any user of the fence to reacquire that fence before continuing with their access. One use is during GPU reset where the fence register is lost and we need to revoke concurrent userspace access via GTT mmaps until the hardware has been reset and the fence registers have been restored.

void **i915_gem_restore_fences**(struct drm_i915_private * *dev_priv*)
restore fence state

Parameters

struct drm_i915_private * dev_priv i915 device private

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 **i915_gem_detect_bit_6_swizzle**(struct drm_i915_private * *dev_priv*)
detect bit 6 swizzling pattern

Parameters

struct drm_i915_private * dev_priv i915 device private

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 mmaps (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.

7.3.6 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 invovlement.

7.3.7 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*, u64 *min_size*, u64 *alignment*, unsigned *cache_level*, 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

u64 min_size size of the desired free space

u64 alignment alignment constraint of the desired free space

unsigned cache_level cache_level 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 *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 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*)
Evict all idle vmas from a vm

Parameters

struct i915_address_space * vm Address space to cleanse

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.

7.3.8 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 drm_i915_private * *dev_priv*, unsigned long *target*, unsigned *flags*)
Shrink buffer object caches

Parameters

struct drm_i915_private * dev_priv i915 device

unsigned long target amount of memory to make available, in pages

unsigned flags 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 to 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 * *dev_priv*)
Shrink buffer object caches completely

Parameters

struct drm_i915_private * dev_priv i915 device

Description

This is a simple wraper 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_shrinker_init**(struct drm_i915_private * *dev_priv*)
Initialize i915 shrinker

Parameters

struct drm_i915_private * dev_priv i915 device

Description

This function registers and sets up the i915 shrinker and OOM handler.

void **i915_gem_shrinker_cleanup**(struct drm_i915_private * *dev_priv*)
Clean up i915 shrinker

Parameters

struct drm_i915_private * dev_priv i915 device

Description

This function unregisters the i915 shrinker and OOM handler.

7.4 GuC

7.4.1 GuC-specific firmware loader

intel_guc: Top level structure of guc. It handles firmware loading and manages client pool and doorbells. intel_guc owns a i915_guc_client to replace the legacy ExecList submission.

Firmware versioning: The firmware build process will generate a version header file with major and minor version defined. The versions are built into CSS header of firmware. i915 kernel driver set the minimal firmware version required per platform. The firmware installation package will install (symbolic link) proper version of firmware.

GuC address space: GuC does not allow any gfx GGTT address that falls into range [0, WOPCM_TOP), which is reserved for Boot ROM, SRAM and WOPCM. Currently this top address is 512K. In order to exclude 0-512K address space from GGTT, all gfx objects used by GuC is pinned with PIN_OFFSET_BIAS along with size of WOPCM.

int **intel_guc_init_hw**(struct intel_guc * *guc*)
finish preparing the GuC for activity

Parameters

struct intel_guc * guc intel_guc structure

Description

Called during driver loading and also after a GPU reset.

The main action required here it to load the GuC uCode into the device. The firmware image should have already been fetched into memory by the earlier call to `intel_guc_init()`, so here we need only check that worked, and then transfer the image to the h/w.

Return

non-zero code on error

int **intel_guc_select_fw**(struct intel_guc * *guc*)
selects GuC firmware for loading

Parameters

struct intel_guc * guc intel_guc struct

Return

zero when we know firmware, non-zero in other case

7.4.2 GuC-based command submission

GuC client: A i915_guc_client refers to a submission path through GuC. Currently, there is only one of these (the execbuf_client) and this one is charged with all submissions to the GuC. This struct is the owner of a doorbell, a process descriptor and a workqueue (all of them inside a single gem object that contains all required pages for these elements).

GuC stage descriptor: During initialization, the driver allocates a static pool of 1024 such descriptors, and shares them with the GuC. Currently, there exists a 1:1 mapping between a i915_guc_client and a guc_stage_desc (via the client's stage_id), so effectively only one gets used. This stage descriptor lets the GuC know about the doorbell, workqueue and process descriptor. Theoretically, it also lets the GuC know about our HW contexts (context ID, etc...), but we actually employ a kind of submission where the GuC uses the LRCA sent via the work item instead (the single guc_stage_desc associated to execbuf client contains information about the default kernel context only, but this is essentially unused). This is called a "proxy" 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. See `intel_guc_send()`

Doorbells: Doorbells are interrupts to uKernel. A doorbell is a single cache line (QW) mapped into process space.

Work Items: There are several types of work items that the host may place into a workqueue, each with its own requirements and limitations. Currently only WQ_TYPE_INORDER is needed to support legacy submission via GuC, which represents in-order queue. The kernel driver packs ring tail pointer and an ELSP context descriptor dword into Work Item. See `guc_wq_item_append()`

ADS: The Additional Data Struct (ADS) has pointers for different buffers used by the GuC. One single gem object contains the ADS struct itself (guc_ads), the scheduling policies (guc_policies), a structure describing a collection of register sets (guc_mmio_reg_state) and some extra pages for the GuC to save its internal state for sleep.

int **i915_guc_wq_reserve**(struct drm_i915_gem_request * *request*)
reserve space in the GuC's workqueue

Parameters

struct drm_i915_gem_request * request request associated with the commands

Return

0 if space is available -EAGAIN if space is not currently available

This function must be called (and must return 0) before a request is submitted to the GuC via `i915_guc_submit()` below. Once a result of 0 has been returned, it must be balanced by a corresponding call to `submit()`.

Reservation allows the caller to determine in advance that space will be available for the next submission before committing resources to it, and helps avoid late failures with complicated recovery paths.

void **__i915_guc_submit**(struct drm_i915_gem_request * *rq*)
Submit commands through GuC

Parameters

struct drm_i915_gem_request * rq request associated with the commands

Description

The caller must have already called *i915_guc_wq_reserve()* above with a result of 0 (success), guaranteeing that there is space in the work queue for the new request, so enqueuing the item cannot fail.

Bad Things Will Happen if the caller violates this protocol e.g. calls `submit()` when `_reserve()` says there's no space, or calls `_submit()` a different number of times from (successful) calls to `_reserve()`.

The only error here arises if the doorbell hardware isn't functioning as expected, which really shouln't happen.

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_WOPCM_TOP) because that range is reserved inside GuC.

Return

A i915_vma if successful, otherwise an ERR_PTR.

struct i915_guc_client * **guc_client_alloc**(struct drm_i915_private * *dev_priv*, uint32_t *engines*, uint32_t *priority*, struct i915_gem_context * *ctx*)
Allocate an i915_guc_client

Parameters

struct drm_i915_private * dev_priv driver private data structure

uint32_t engines The set of engines to enable for this client

uint32_t priority four levels priority _CRITICAL, _HIGH, _NORMAL and _LOW The kernel client to replace ExecList submission is created with NORMAL priority. Priority of a client for scheduler can be HIGH, while a preemption context can use CRITICAL.

struct i915_gem_context * ctx the context that owns the client (we use the default render context)

Return

An i915_guc_client object if success, else NULL.

int **intel_guc_suspend**(struct drm_i915_private * *dev_priv*)
notify GuC entering suspend state

Parameters

struct drm_i915_private * dev_priv i915 device private

int **intel_guc_resume**(struct drm_i915_private * *dev_priv*)
notify GuC resuming from suspend state

Parameters

struct drm_i915_private * dev_priv i915 device private

7.4.3 GuC Firmware Layout

The GuC firmware layout looks like this:

uc_css_header contains major/minor version
uCode
RSA signature
modulus key
exponent val

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.

HuC firmware layout is same as GuC firmware.

HuC firmware css header is different. However, the only difference is where the version information is saved. The uc_css_header is unified to support both. Driver should get HuC version from uc_css_header.huc_sw_version, while uc_css_header.guc_sw_version for GuC.

7.5 Tracing

This sections covers all things related to the tracepoints implemented in the i915 driver.

7.5.1 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.

7.5.2 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.

7.5.3 switch_mm

This tracepoint allows tracking of the mm switch, which is an important point in the lifetime of the vm in the legacy submission path. This tracepoint is called only if full ppgtt is enabled.

7.6 Perf

7.6.1 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.

7.6.2 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.

7.6.3 i915 Driver Entry Points

This section covers the entrypoints exported outside of i915_perf.c to integrate with drm/i915 and to handle the *DRM_I915_PERF_OPEN* ioctl.

void **i915_perf_init**(struct drm_i915_private * *dev_priv*)
  initialize i915-perf state on module load

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*)
  Counter part to *i915_perf_init()*

Parameters

struct drm_i915_private * dev_priv i915 device instance

void **i915_perf_register**(struct drm_i915_private * *dev_priv*)
  exposes i915-perf to userspace

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*)
  hide i915-perf from userspace

Parameters

struct drm_i915_private * dev_priv 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_sem.

Most of the implementation details are handled by *i915_perf_open_ioctl_locked()* after taking the `drm_i915_private->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.

7.6.4 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 drm_i915_private * dev_priv;
  struct list_head link;
  u32 sample_flags;
  int sample_size;
  struct i915_gem_context * ctx;
  bool enabled;
  const struct i915_perf_stream_ops * ops;
};
```

Members

dev_priv i915 drm device

link Links the stream into `:c:type:`drm_i915_private->streams <drm_i915_private>``

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.

ops The callbacks providing the implementation of this specific type of configured stream.

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 drm_i915_private * *dev_priv*, u64 __user * *uprops*, u32 *n_props*, struct *perf_open_properties* * *props*)
validate + copy userspace stream open properties

Parameters

struct drm_i915_private * dev_priv i915 device 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 drm_i915_private * *dev_priv*, 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 drm_i915_private * dev_priv i915 device 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 `drm_i915_private->perf.lock` mutex taken to serialize with any non-file-operation driver hooks.

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.

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.

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 `drm_i915_private->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).

unsigned int **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

unsigned int **i915_perf_poll_locked**(struct drm_i915_private * *dev_priv*, struct *i915_perf_stream* * *stream*, struct file * *file*, poll_table * *wait*)

poll_wait() with a suitable wait queue for stream

Parameters

struct drm_i915_private * dev_priv i915 device instance

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.

Note

The drm_i915_private->perf.lock mutex has been taken to serialize with any non-file-operation driver hooks.

Return

any poll events that are ready without sleeping

7.6.5 i915 Perf Observation Architecture Stream

struct **i915_oa_ops**

Gen specific implementation of an OA unit stream

Definition

```
struct i915_oa_ops {
  void (* init_oa_buffer) (struct drm_i915_private *dev_priv);
  int (* select_metric_set) (struct drm_i915_private *dev_priv);
  int (* enable_metric_set) (struct drm_i915_private *dev_priv);
  void (* disable_metric_set) (struct drm_i915_private *dev_priv);
  void (* oa_enable) (struct drm_i915_private *dev_priv);
  void (* oa_disable) (struct drm_i915_private *dev_priv);
  int (* read) (struct i915_perf_stream *stream,char __user *buf,size_t count, size_t *offset);
  u32 (* oa_hw_tail_read) (struct drm_i915_private *dev_priv);
};
```

Members

init_oa_buffer Resets the head and tail pointers of the circular buffer for periodic OA reports.

Called when first opening a stream for OA metrics, but also may be called in response to an OA buffer overflow or other error condition.

Note it may be necessary to clear the full OA buffer here as part of maintaining the invariable that new reports must be written to zeroed memory for us to be able to reliable detect if an expected report has not yet landed in memory. (At least on Haswell the OA buffer tail pointer is not synchronized with reports being visible to the CPU)

select_metric_set The auto generated code that checks whether a requested OA config is applicable to the system and if so sets up the mux, oa and flex eu register config pointers according to the current dev_priv->perf.oa.metrics_set.

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.

7.6.6 All i915 Perf Internals

This section simply includes all 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 ctx_handle;
  int metrics_set;
  int oa_format;
  bool oa_periodic;
  int oa_period_exponent;
};
```

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

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

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 drm_i915_private * *dev_priv*)
check for data and update tail ptr state

Parameters

struct drm_i915_private * dev_priv i915 device 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 has the side effect of updating the oa_buffer.tails[], .aging_timestamp and .aged_tail_idx state used for reading.

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*)

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*)

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 **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.

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

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.

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_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.

ssize_t **i915_perf_read_locked**(struct *i915_perf_stream* * *stream*, struct file * *file*, char __user * *buf*, size_t *count*, loff_t * *ppos*)
i915_perf_stream_ops->read with error normalisation

Parameters

struct i915_perf_stream * stream An i915 perf stream

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

Besides wrapping *i915_perf_stream_ops->read* this provides a common place to ensure that if we've successfully copied any data then reporting that takes precedence over any internal error status, so the data isn't lost.

For example ret will be -ENOSPC whenever there is more buffered data than can be copied to userspace, but that's only interesting if we weren't able to copy some data because it implies the userspace buffer is too small to receive a single record (and we never split records).

Another case with ret == -EFAULT is more of a grey area since it would seem like bad form for userspace to ask us to overrun its buffer, but the user knows best:

http://yarchive.net/comp/linux/partial_reads_writes.html

Return

The number of bytes copied or a negative error code on failure.

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.

unsigned int **i915_perf_poll_locked**(struct drm_i915_private * *dev_priv*, struct *i915_perf_stream* * *stream*, struct file * *file*, poll_table * *wait*)
`poll_wait()` with a suitable wait queue for stream

Parameters

struct drm_i915_private * dev_priv i915 device instance

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.

Note

The `drm_i915_private->perf`.lock mutex has been taken to serialize with any non-file-operation driver hooks.

Return

any poll events that are ready without sleeping

unsigned int **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

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).

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

Note

The `drm_i915_private->perf`.lock mutex has been taken to serialize with any non-file-operation driver hooks.

Return

zero on success or a negative error code. Returns -EINVAL for an unknown ioctl request.

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_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 `drm_i915_private->perf`.lock mutex has been taken to serialize with any non-file-operation driver hooks.

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_open_ioctl_locked**(struct drm_i915_private * *dev_priv*, 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 drm_i915_private * dev_priv i915 device 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 `drm_i915_private->perf.lock` mutex taken to serialize with any non-file-operation driver hooks.

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.

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.

Return

zero on success or a negative error code.

int **read_properties_unlocked**(struct drm_i915_private * *dev_priv*, u64 __user * *uprops*, u32 *n_props*, struct *perf_open_properties* * *props*)

validate + copy userspace stream open properties

Parameters

struct drm_i915_private * dev_priv i915 device 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**(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_sem.

Most of the implementation details are handled by *i915_perf_open_ioctl_locked()* after taking the `drm_i915_private->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.

void **i915_perf_register**(struct drm_i915_private * *dev_priv*)
exposes i915-perf to userspace

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*)
hide i915-perf from userspace

Parameters

struct drm_i915_private * dev_priv 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()*.

void **i915_perf_init**(struct drm_i915_private * *dev_priv*)
initialize i915-perf state on module load

Parameters

struct drm_i915_private * dev_priv 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 * *dev_priv*)
Counter part to *i915_perf_init()*

Parameters

struct drm_i915_private * dev_priv i915 device instance

CHAPTER

EIGHT

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

8.1 Video Processing Unit

The Amlogic Meson Display controller is composed of several components that are going to be documented below:

```
DMC|---------------VPU (Video Processing Unit)----------------|------HHI------|
   | vd1   _______     _____________    _________________     |               |
D  |-------|      |----|            |   |                |    |   HDMI PLL    |
D  | vd2   | VIU  |    | Video Post |   | Video Encoders |<---|-----VCLK      |
R  |-------|      |----| Processing |   |                |    |               |
   | osd2  |      |    |            |---| Enci ----------|----|-----VDAC------|
R  |-------| CSC  |----| Scalers    |   | Encp ----------|----|----HDMI-TX----|
A  | osd1  |      |    | Blenders   |   | Encl ----------|----|---------------|
M  |-------|______|----|____________|   |________________|    |               |
___|__________________________________________________________|_______________|
```

8.2 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

8.3 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

8.4 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 :

```
       _____   _____   ____________________
vd1---|     |-|     | | VENC     /---------|----VDAC
vd2---| VIU |-| VPP |-|-----ENCI/-ENCI_DVI-|-|
osd1--|     |-|     | | \                  | X--HDMI-TX
osd2--|_____|-|_____| |  |\-ENCP--ENCP_DVI-|-|
                      |  |                 |
                      |  \--ENCL-----------|----LVDS
                      |____________________|
```

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 initialy desined 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.

8.5 Video Canvas Management

CANVAS is a memory zone where physical memory frames information are stored for the VIU to scanout.

8.6 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 :

```
 __________   _________            _____
|          | |         |          |     |--ENCI
| HDMI PLL |-| PLL_DIV |--- VCLK--|     |--ENCL
|__________| |_________| \        | MUX |--ENCP
                          --VCLK2-|     |--VDAC
                                  |_____|--HDMI-TX
```

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.

8.7 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

```
 ___________________________________
|            HDMI TOP               |<= HPD
|___________________________________|
|                  |                |
|  Synopsys HDMI   |   HDMI PHY     |=> TMDS
```

```
|    Controller    |_________________|
|_____________________________________|<=> DDC
```

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

CHAPTER

NINE

DRM/PL111 ARM PRIMECELL PL111 CLCD DRIVER

The PL111 is a simple LCD controller that can support TFT and STN displays. This driver exposes a standard KMS interface for them.

This driver uses the same Device Tree binding as the fbdev CLCD driver. While the fbdev driver supports panels that may be connected to the CLCD internally to the CLCD driver, in DRM the panels get split out to drivers/gpu/drm/panels/. This means that, in converting from using fbdev to using DRM, you also need to write a panel driver (which may be as simple as an entry in panel-simple.c).

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 X11 software cursor layer.

TODO:

- Fix race between setting plane base address and getting IRQ for vsync firing the pageflip completion.
- Expose the correct set of formats we can support based on the "arm,pl11x,tft-r0g0b0-pads" DT property.
- Use the "max-memory-bandwidth" DT property to filter the supported formats.
- 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.

CHAPTER

TEN

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.

10.1 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.

10.1.1 Host1x Infrastructure Reference

struct **host1x_client_ops**
    host1x client operations

Definition

```
struct host1x_client_ops {
  int (* init) (struct host1x_client *client);
  int (* exit) (struct host1x_client *client);
};
```

Members

init host1x client initialization code

exit host1x client tear down code

struct **host1x_client**
host1x client structure

Definition

```
struct host1x_client {
  struct list_head list;
  struct device * parent;
  struct device * dev;
  const struct host1x_client_ops * ops;
  enum host1x_class class;
  struct host1x_channel * channel;
  struct host1x_syncpt ** syncpts;
  unsigned int num_syncpts;
};
```

Members

list list node for the host1x client

parent pointer to struct device representing the host1x controller

dev pointer to struct device backing this host1x client

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

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 tho 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.

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.

int **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.

10.1.2 Host1x Syncpoint Reference

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 device * *dev*, unsigned long *flags*)
request a syncpoint

Parameters

struct device * dev device 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_free()`.

void **host1x_syncpt_free**(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. Note that client drivers must ensure that the syncpoint doesn't remain under the control of hardware after calling this function, otherwise two clients may end up trying to access the same syncpoint concurrently.

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**(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_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

10.2 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.

10.2.1 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 CRTC 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.

10.2.2 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.

10.3 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.

10.3.1 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.

10.3.2 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.

10.3.3 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.

CHAPTER

ELEVEN

DRM/TINYDRM DRIVER LIBRARY

This library provides driver helpers for very simple display hardware.

It is based on *drm_simple_display_pipe* coupled with a *drm_connector* which has only one fixed *drm_display_mode*. The framebuffers are backed by the cma helper and have support for framebuffer flushing (dirty). fbdev support is also included.

11.1 Core functionality

The driver allocates *tinydrm_device*, initializes it using *devm_tinydrm_init()*, sets up the pipeline using *tinydrm_display_pipe_init()* and registers the DRM device using *devm_tinydrm_register()*.

struct **tinydrm_device**
tinydrm device

Definition

```
struct tinydrm_device {
  struct drm_device * drm;
  struct drm_simple_display_pipe pipe;
  struct mutex dirty_lock;
  struct drm_fbdev_cma * fbdev_cma;
  struct drm_atomic_state * suspend_state;
  const struct drm_framebuffer_funcs * fb_funcs;
};
```

Members

drm DRM device

pipe Display pipe structure

dirty_lock Serializes framebuffer flushing

fbdev_cma CMA fbdev structure

suspend_state Atomic state when suspended

fb_funcs Framebuffer functions used when creating framebuffers

TINYDRM_GEM_DRIVER_OPS()
default tinydrm gem operations

Parameters

Description

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

TINYDRM_MODE(*hd*, *vd*, *hd_mm*, *vd_mm*)
tinydrm 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 creates a *drm_display_mode* for use with tinydrm.

void **tinydrm_lastclose**(struct drm_device * *drm*)
DRM lastclose helper

Parameters

struct drm_device * drm DRM device

Description

This function ensures that fbdev is restored when drm_lastclose() is called on the last *drm_release()*. Drivers can use this as their *drm_driver->lastclose* callback.

struct *drm_gem_object* * **tinydrm_gem_cma_prime_import_sg_table**(struct drm_device * *drm*, struct dma_buf_attachment * *attach*, struct sg_table * *sgt*)
Produce a CMA GEM object from another driver's scatter/gather table of pinned pages

Parameters

struct drm_device * drm DRM 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 using *drm_gem_cma_prime_import_sg_table()*. It sets the kernel virtual address on the CMA object. Drivers should use this as their *drm_driver->gem_prime_import_sg_table* callback if they need the virtual address. *tinydrm_gem_cma_free_object()* should be used in combination with this function.

Return

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

void **tinydrm_gem_cma_free_object**(struct *drm_gem_object* * *gem_obj*)
Free resources associated with a CMA GEM object

Parameters

struct drm_gem_object * gem_obj GEM object to free

Description

This function frees the backing memory of the CMA GEM object, cleans up the GEM object state and frees the memory used to store the object itself using *drm_gem_cma_free_object()*. It also handles PRIME buffers which has the kernel virtual address set by *tinydrm_gem_cma_prime_import_sg_table()*. Drivers can use this as their *drm_driver->gem_free_object* callback.

int **devm_tinydrm_init**(struct device * *parent*, struct *tinydrm_device* * *tdev*, const struct *drm_framebuffer_funcs* * *fb_funcs*, struct *drm_driver* * *driver*)
Initialize tinydrm device

Parameters

struct device * parent Parent device object

struct tinydrm_device * tdev tinydrm device

const struct drm_framebuffer_funcs * fb_funcs Framebuffer functions

struct drm_driver * driver DRM driver

Description

This function initializes **tdev**, the underlying DRM device and it's mode_config. Resources will be automatically freed on driver detach (devres) using *drm_mode_config_cleanup()* and *drm_dev_unref()*.

Return

Zero on success, negative error code on failure.

int **devm_tinydrm_register**(struct *tinydrm_device* * *tdev*)
Register tinydrm device

Parameters

struct tinydrm_device * tdev tinydrm device

Description

This function registers the underlying DRM device and fbdev. These resources will be automatically unregistered on driver detach (devres) and the display pipeline will be disabled.

Return

Zero on success, negative error code on failure.

void **tinydrm_shutdown**(struct *tinydrm_device* * *tdev*)
Shutdown tinydrm

Parameters

struct tinydrm_device * tdev tinydrm device

Description

This function makes sure that the display pipeline is disabled. Used by drivers in their shutdown callback to turn off the display on machine shutdown and reboot.

int **tinydrm_suspend**(struct *tinydrm_device* * *tdev*)
Suspend tinydrm

Parameters

struct tinydrm_device * tdev tinydrm device

Description

Used in driver PM operations to suspend tinydrm. Suspends fbdev and DRM. Resume with *tinydrm_resume()*.

Return

Zero on success, negative error code on failure.

int **tinydrm_resume**(struct *tinydrm_device* * *tdev*)
Resume tinydrm

Parameters

struct tinydrm_device * tdev tinydrm device

Description

Used in driver PM operations to resume tinydrm. Suspend with *tinydrm_suspend()*.

Return

Zero on success, negative error code on failure.

void **tinydrm_display_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 does a full framebuffer flush if the plane framebuffer has changed. It also handles vblank events. Drivers can use this as their *drm_simple_display_pipe_funcs->update* callback.

int **tinydrm_display_pipe_prepare_fb**(struct *drm_simple_display_pipe* * *pipe*, struct *drm_plane_state* * *plane_state*)
Display pipe prepare_fb helper

Parameters

struct drm_simple_display_pipe * pipe Simple display pipe

struct drm_plane_state * plane_state Plane state

Description

This function uses *drm_fb_cma_prepare_fb()* to check if the plane FB has an dma-buf attached, extracts the exclusive fence and attaches it to plane state for the atomic helper to wait on. Drivers can use this as their *drm_simple_display_pipe_funcs->prepare_fb* callback.

int **tinydrm_display_pipe_init**(struct *tinydrm_device* * *tdev*, const struct *drm_simple_display_pipe_funcs* * *funcs*, int *connector_type*, const uint32_t * *formats*, unsigned int *format_count*, const struct *drm_display_mode* * *mode*, unsigned int *rotation*)
Initialize display pipe

Parameters

struct tinydrm_device * tdev tinydrm device

const struct drm_simple_display_pipe_funcs * funcs Display pipe functions

int connector_type Connector type

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 Supported mode

unsigned int rotation Initial **mode** 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**.

Return

Zero on success, negative error code on failure.

11.2 Additional helpers

bool **tinydrm_machine_little_endian**(void)
Machine is little endian

Parameters

void no arguments

Return

true if *defined(__LITTLE_ENDIAN)*, false otherwise

void **tinydrm_dbg_spi_message**(struct spi_device * *spi*, struct spi_message * *m*)
Dump SPI message

Parameters

struct spi_device * spi SPI device

struct spi_message * m SPI message

Description

Dumps info about the transfers in a SPI message including buffer content. DEBUG has to be defined for this function to be enabled alongside setting the DRM_UT_DRIVER bit of drm_debug.

bool **tinydrm_merge_clips**(struct drm_clip_rect * *dst*, struct drm_clip_rect * *src*, unsigned int *num_clips*, unsigned int *flags*, u32 *max_width*, u32 *max_height*)
Merge clip rectangles

Parameters

struct drm_clip_rect * dst Destination clip rectangle

struct drm_clip_rect * src Source clip rectangle(s)

unsigned int num_clips Number of **src** clip rectangles

unsigned int flags Dirty fb ioctl flags

u32 max_width Maximum width of **dst**

u32 max_height Maximum height of **dst**

Description

This function merges **src** clip rectangle(s) into **dst**. If **src** is NULL, **max_width** and **min_width** is used to set a full **dst** clip rectangle.

Return

true if it's a full clip, false otherwise

void **tinydrm_memcpy**(void * *dst*, void * *vaddr*, struct *drm_framebuffer* * *fb*, struct drm_clip_rect * *clip*)
Copy clip buffer

Parameters

void * dst Destination buffer

void * vaddr Source buffer

struct drm_framebuffer * fb DRM framebuffer

struct drm_clip_rect * clip Clip rectangle area to copy

void **tinydrm_swab16**(u16 * *dst*, void * *vaddr*, struct *drm_framebuffer* * *fb*, struct drm_clip_rect * *clip*)
Swap bytes into clip buffer

Parameters

u16 * dst RGB565 destination buffer

void * vaddr RGB565 source buffer

struct drm_framebuffer * fb DRM framebuffer

struct drm_clip_rect * clip Clip rectangle area to copy

void **tinydrm_xrgb8888_to_rgb565**(u16 * *dst*, void * *vaddr*, struct *drm_framebuffer* * *fb*, struct drm_clip_rect * *clip*, bool *swap*)
Convert XRGB8888 to RGB565 clip buffer

Parameters

u16 * dst RGB565 destination buffer

void * vaddr XRGB8888 source buffer

struct drm_framebuffer * fb DRM framebuffer

struct drm_clip_rect * clip Clip rectangle area to copy

bool swap Swap bytes

Description

Drivers can use this function for RGB565 devices that don't natively support XRGB8888.

struct backlight_device * **tinydrm_of_find_backlight**(struct device * *dev*)
Find backlight device in device-tree

Parameters

struct device * dev Device

Description

This function looks for a DT node pointed to by a property named 'backlight' and uses `of_find_backlight_by_node()` to get the backlight device. Additionally if the brightness property is zero, it is set to max_brightness.

Return

NULL if there's no backlight property. Error pointer -EPROBE_DEFER if the DT node is found, but no backlight device is found. If the backlight device is found, a pointer to the structure is returned.

int **tinydrm_enable_backlight**(struct backlight_device * *backlight*)
Enable backlight helper

Parameters

struct backlight_device * backlight Backlight device

Return

Zero on success, negative error code on failure.

int **tinydrm_disable_backlight**(struct backlight_device * *backlight*)
Disable backlight helper

Parameters

struct backlight_device * backlight Backlight device

Return

Zero on success, negative error code on failure.

size_t **tinydrm_spi_max_transfer_size**(struct spi_device * *spi*, size_t *max_len*)
Determine max SPI transfer size

Parameters

struct spi_device * spi SPI device

size_t max_len Maximum buffer size needed (optional)

Description

This function returns the maximum size to use for SPI transfers. It checks the SPI master, the optional **max_len** and the module parameter spi_max and returns the smallest.

Return

Maximum size for SPI transfers

bool **tinydrm_spi_bpw_supported**(struct spi_device * *spi*, u8 *bpw*)
Check if bits per word is supported

Parameters

struct spi_device * spi SPI device

u8 bpw Bits per word

Description

This function checks to see if the SPI master driver supports **bpw**.

Return

True if **bpw** is supported, false otherwise.

int **tinydrm_spi_transfer**(struct spi_device * *spi*, u32 *speed_hz*, struct spi_transfer * *header*, u8 *bpw*, const void * *buf*, size_t *len*)
SPI transfer helper

Parameters

struct spi_device * spi SPI device

u32 speed_hz Override speed (optional)

struct spi_transfer * header Optional header transfer

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 master driver can handle. If the machine is Little Endian and the SPI master driver doesn't support 16 bits per word, it swaps the bytes and does a 8-bit transfer. If **header** is set, it is prepended to each SPI message.

Return

Zero on success, negative error code on failure.

11.3 MIPI DBI Compatible Controllers

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:

1. Motorola 6800 type parallel bus
2. Intel 8080 type parallel bus
3. SPI type with 3 options:
   (a) 9-bit with the Data/Command signal as the ninth bit
   (b) Same as above except it's sent as 16 bits

(c) 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 controller

Definition

```
struct mipi_dbi {
  struct tinydrm_device tinydrm;
  struct spi_device * spi;
  bool enabled;
  struct mutex cmdlock;
  int (* command) (struct mipi_dbi *mipi, u8 cmd, u8 *param, size_t num);
  const u8 * read_commands;
  struct gpio_desc * dc;
  u16 * tx_buf;
  void * tx_buf9;
  size_t tx_buf9_len;
  bool swap_bytes;
  struct gpio_desc * reset;
  unsigned int rotation;
  struct backlight_device * backlight;
  struct regulator * regulator;
};
```

Members

tinydrm tinydrm base

spi SPI device

enabled Pipeline is enabled

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.

dc Optional D/C gpio.

tx_buf Buffer used for transfer (copy clip rect area)

tx_buf9 Buffer used for Option 1 9-bit conversion

tx_buf9_len Size of tx_buf9.

swap_bytes Swap bytes in buffer before transfer

reset Optional reset gpio

rotation initial rotation in degrees Counter Clock Wise

backlight backlight device (optional)

regulator power regulator (optional)

mipi_dbi_command(*mipi*, *cmd*, *seq...*)
MIPI DCS command with optional parameter(s)

Parameters

mipi MIPI 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.

int **mipi_dbi_command_read**(struct *mipi_dbi* * *mipi*, u8 *cmd*, u8 * *val*)
MIPI DCS read command

Parameters

struct mipi_dbi * mipi MIPI 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* * *mipi*, u8 *cmd*, u8 * *data*, size_t *len*)
MIPI DCS command with parameter(s) in an array

Parameters

struct mipi_dbi * mipi MIPI structure

u8 cmd Command

u8 * data Parameter buffer

size_t len Buffer length

Return

Zero on success, negative error code on failure.

void **mipi_dbi_pipe_enable**(struct *drm_simple_display_pipe* * *pipe*, struct *drm_crtc_state* * *crtc_state*)
MIPI DBI pipe enable helper

Parameters

struct drm_simple_display_pipe * pipe Display pipe

struct drm_crtc_state * crtc_state CRTC state

Description

This function enables backlight. Drivers can use this as their *drm_simple_display_pipe_funcs->enable* callback.

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 or if not the display memory is blanked. Drivers can use this as their *drm_simple_display_pipe_funcs->disable* callback.

int **mipi_dbi_init**(struct device * *dev*, struct *mipi_dbi* * *mipi*, const struct *drm_simple_display_pipe_funcs* * *pipe_funcs*, struct *drm_driver* * *driver*, const struct *drm_display_mode* * *mode*, unsigned int *rotation*)
MIPI DBI initialization

Parameters

struct device * dev Parent device

struct mipi_dbi * mipi *mipi_dbi* structure to initialize

const struct drm_simple_display_pipe_funcs * pipe_funcs Display pipe functions

struct drm_driver * driver DRM driver

const struct drm_display_mode * mode Display mode

unsigned int rotation Initial rotation in degrees Counter Clock Wise

Description

This function initializes a *mipi_dbi* structure and it's underlying **tinydrm_device**. It also sets up the display pipeline.

Supported formats: Native RGB565 and emulated XRGB8888.

Objects created by this function will be automatically freed on driver detach (devres).

Return

Zero on success, negative error code on failure.

void **mipi_dbi_hw_reset**(struct *mipi_dbi* * *mipi*)
Hardware reset of controller

Parameters

struct mipi_dbi * mipi MIPI DBI structure

Description

Reset controller if the *mipi_dbi->reset* gpio is set.

bool **mipi_dbi_display_is_on**(struct *mipi_dbi* * *mipi*)
Check if display is on

Parameters

struct mipi_dbi * mipi 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_spi_init**(struct spi_device * *spi*, struct *mipi_dbi* * *mipi*, struct gpio_desc * *dc*, const struct *drm_simple_display_pipe_funcs* * *pipe_funcs*, struct *drm_driver* * *driver*, const struct *drm_display_mode* * *mode*, unsigned int *rotation*)
Initialize MIPI DBI SPI interfaced controller

Parameters

struct spi_device * spi SPI device

struct mipi_dbi * mipi *mipi_dbi* structure to initialize

struct gpio_desc * dc D/C gpio (optional)

const struct drm_simple_display_pipe_funcs * pipe_funcs Display pipe functions

struct drm_driver * driver DRM driver

const struct drm_display_mode * mode Display mode

unsigned int rotation Initial rotation in degrees Counter Clock Wise

Description

This function sets *mipi_dbi->command*, enables mipi->read_commands for the usual read commands and initializes **mipi** using *mipi_dbi_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_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.

Return

Zero on success, negative error code on failure.

CHAPTER

TWELVE

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.

12.1 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.

12.1.1 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.

12.1.2 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.

12.1.3 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.

12.1.4 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.

12.1.5 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.

12.1.6 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.

12.1.7 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.

12.2 Memory Management and 3D Command Submission

This section covers the GEM implementation in the vc4 driver.

12.2.1 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 CMA helper functions to allocate contiguous ranges of physical memory for our BOs.

Since the CMA 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.

12.2.2 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.

12.2.3 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.

12.2.4 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 mmaps, GEM prime export, or rendering to from a CL), so this validation is only performed at BO creation time.

12.2.5 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.

CHAPTER

THIRTEEN

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.

13.1 Modes of Use

13.1.1 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.

13.1.2 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.

When the driver decides to power up or down, it notifies vga_switcheroo thereof so that it can (a) power the audio device on the GPU up or down, and (b) update its internal power state representation for the device. This is achieved by *vga_switcheroo_set_dynamic_switch()*.

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 *vga_switcheroo_init_domain_pm_optimus_hdmi_audio()*, which augments the audio device's resume function.

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

13.2 API

13.2.1 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*, enum *vga_switcheroo_client_id* *id*)
register audio client

Parameters

struct pci_dev * pdev client pci device

const struct vga_switcheroo_client_ops * ops client callbacks

enum vga_switcheroo_client_id id client identifier

Description

Register audio client (audio device on a GPU). 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.

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.

void **vga_switcheroo_set_dynamic_switch**(struct pci_dev * *pdev*, enum *vga_switcheroo_state* *dynamic*)
helper for driver power control

Parameters

struct pci_dev * pdev client pci device

enum vga_switcheroo_state dynamic new power state

Description

Helper for GPUs whose power state is controlled by the driver's runtime pm. When the driver decides to power up or down, it notifies vga_switcheroo thereof using this helper so that it can (a) power the audio device on the GPU up or down, and (b) update its internal power state representation for the device.

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.

int **vga_switcheroo_init_domain_pm_optimus_hdmi_audio**(struct device * *dev*, struct dev_pm_domain * *domain*)
helper for driver power control

Parameters

struct device * dev audio client device

struct dev_pm_domain * domain power domain

Description

Helper for GPUs whose power state is controlled by the driver's runtime pm. When the audio device resumes, the GPU needs to be woken. This helper augments the audio device's resume function to do that.

Return

0 on success, -EINVAL if no power management operations are defined for this device.

13.2.2 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);
};
```

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

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.

13.2.3 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.

13.2.4 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;
  struct dentry * switch_file;
  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

switch_file file 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;
};
```

Members

pdev client pci device

fb_info framebuffer to which console is remapped on switching

pwr_state current power 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

Description

Registered client. A client can be either a GPU or an audio device on a GPU. For audio clients, the **fb_info**, **active** and **driver_power_control** members are bogus.

13.3 Handlers

13.3.1 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 retinas.

(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 retina gmux are infixed `_index_`.

gmux is also connected to a GPIO pin of the southbridge and thereby is able to trigger an ACPI GPE. On the MBP5 2008/09 it's GPIO pin 22 of the Nvidia MCP79, on all following generations it's GPIO pin 6 of the Intel PCH. The GPE merely signals that an interrupt occurred, the actual type of event is identified by reading a gmux register.

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 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. All newer models use a TI LP8545.

Public functions

bool **apple_gmux_present**(void)
detect if gmux is built into the machine

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 is present and the kernel was configured with CONFIG_APPLE_GMUX, `false` otherwise.

CHAPTER

FOURTEEN

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.

14.1 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.

14.2 In-kernel interface

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

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 **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 archs 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 theirs 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_get**(struct pci_dev * *pdev*, unsigned int *rsrc*, int *interruptible*)
acquire & locks 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

This function acquires VGA resources for the given card and mark those resources locked. If the resource 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()*.

Return

0 on success, negative error code on failure.

int **vga_tryget**(struct pci_dev * *pdev*, unsigned int *rsrc*)
try to acquire & lock legacy VGA resources

Parameters

struct pci_dev * pdev pci devivce of VGA card or NULL for system default

unsigned int rsrc bit mask of resources to acquire and lock

Description

This function performs the same operation as *vga_get()*, but will return an error (-EBUSY) instead of blocking if the resources are already locked by another card. It can be called in any context

On success, release the VGA resource again with *vga_put()*.

Return

0 on success, negative error code on failure.

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 but mask of resource to release

Description

This fuction releases resources previously locked by *vga_get()* or *vga_tryget()*. The resources aren't disabled right away, so that a subsequence *vga_get()* on the same card will succeed immediately. Resources have a counter, so locks are only released if the counter reaches 0.

int **vga_client_register**(struct pci_dev * *pdev*, void * *cookie*, void (*irq_set_state) (void **cookie*, bool *state*, unsigned int (*set_vga_decode) (void **cookie*, bool *decode*)
register or unregister a VGA arbitration client

Parameters

struct pci_dev * pdev pci device of the VGA client

void * cookie client cookie to be used in callbacks

void (*)(void *cookie,bool state) irq_set_state irq state change callback

unsigned int (*)(void *cookie,bool decode) set_vga_decode vga decode change callback

Description

Clients have two callback mechanisms they can use.

irq_set_state callback: If a client can't disable its GPUs VGA resources, then we need to be able to ask it to turn off its irqs when we turn off its mem and io decoding.

set_vga_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 some single GPU laptops seem to require ACPI or BIOS access to the VGA registers to control things like backlights 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.

This function does not check whether a client for **pdev** has been registered already.

To unregister just call this function with **irq_set_state** and **set_vga_decode** both set to NULL for the same **pdev** as originally used to register them.

Return

0 on success, -1 on failure

14.3 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);
```

14.4 xf86VGAArbiter (X server implementation)

X server basically wraps all the functions that touch VGA registers somehow.

14.5 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. http://cgit.freedesktop.org/xorg/xserver/commit/?id=4b42448a2388d40f257774fbffdccaea87bd0347

1. http://lists.freedesktop.org/archives/xorg/2005-March/006663.html
2. http://lists.freedesktop.org/archives/xorg/2005-March/006745.html
3. http://lists.freedesktop.org/archives/xorg/2007-October/029507.html

CHAPTER

FIFTEEN

DRM/BRIDGE/DW-HDMI SYNOPSYS DESIGNWARE HDMI CONTROLLER

15.1 Synopsys DesignWare HDMI Controller

This section covers everything related to the Synopsys DesignWare HDMI Controller implemented as a DRM bridge.

15.1.1 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_5X24	V4L2_YCBCR_ENC_601 or V4L2_YCBCR_ENC_709
YCbCr 4:2:0 10bits	MEDIA_BUS_FMT_UYYVYY10_0_5X30	V4L2_YCBCR_ENC_601 or V4L2_YCBCR_ENC_709
YCbCr 4:2:0 12bits	MEDIA_BUS_FMT_UYYVYY12_0_5X36	V4L2_YCBCR_ENC_601 or V4L2_YCBCR_ENC_709
YCbCr 4:2:0 16bits	MEDIA_BUS_FMT_UYYVYY16_0_5X48	V4L2_YCBCR_ENC_601 or V4L2_YCBCR_ENC_709

CHAPTER

SIXTEEN

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.

16.1 Subsystem-wide refactorings

16.1.1 De-midlayer drivers

With the recent `drm_bus` cleanup patches for 3.17 it is no longer required to have a `drm_bus` structure set up. Drivers can directly set up the `drm_device` structure instead of relying on bus methods in `drm_usb.c` and `drm_pci.c`. The goal is to get rid of the driver's `->load` / `->unload` callbacks and open-code the load/unload sequence properly, using the new two-stage `drm_device` setup/teardown.

Once all existing drivers are converted we can also remove those bus support files for USB and platform devices.

All you need is a GPU for a non-converted driver (currently almost all of them, but also all the virtual ones used by KVM, so everyone qualifies).

Contact: Daniel Vetter, Thierry Reding, respective driver maintainers

16.1.2 Switch from reference/unreference to get/put

For some reason DRM core uses `reference`/`unreference` suffixes for refcounting functions, but kernel uses `get`/`put` (e.g. `kref_get/put()`). It would be good to switch over for consistency, and it's shorter. Needs to be done in 3 steps for each pair of functions:

- Create new `get/put` functions, define the old names as compatibility wrappers
- Switch over each file/driver using a cocci-generated spatch.
- Once all users of the old names are gone, remove them.

This way drivers/patches in the progress of getting merged won't break.

Contact: Daniel Vetter

16.1.3 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 atomic and all you need is a GPU for a non-converted driver (again virtual HW drivers for KVM are still all suitable).

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

16.1.4 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, preferrably 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

16.1.5 Implement deferred fbdev setup in the helper

Many (especially embedded drivers) want to delay fbdev setup until there's a real screen plugged in. This is to avoid the dreaded fallback to the low-res fbdev default. Many drivers have a hacked-up (and often broken) version of this, better to do it once in the shared helpers. Thierry has a patch series, but that one needs to be rebased and final polish applied.

Contact: Thierry Reding, Daniel Vetter, driver maintainers

16.1.6 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.

Contact: Daniel Vetter, respective driver maintainers

16.1.7 Better manual-upload support for atomic

This would be especially useful for tinydrm:

- Add a struct drm_rect dirty_clip to drm_crtc_state. When duplicating the crtc state, clear that to the max values, x/y = 0 and w/h = MAX_INT, in __drm_atomic_helper_crtc_duplicate_state().
- Move tinydrm_merge_clips into drm_framebuffer.c, dropping the *tinydrm* prefix ofc and using **drm_fb_**. drm_framebuffer.c makes sense since this is a function useful to implement the fb->dirty function.
- Create a new drm_fb_dirty function which does essentially what e.g. mipi_dbi_fb_dirty does. You can use e.g. drm_atomic_helper_update_plane as the template. But instead of doing a simple full-screen plane update, this new helper also sets crtc_state->dirty_clip to the right coordinates. And of course it needs to check whether the fb is actually active (and maybe where), so there's some book-keeping involved. There's also some good fun involved in scaling things appropriately. For that case we might simply give up and declare the entire area covered by the plane as dirty.

Contact: Noralf Trønnes, Daniel Vetter

16.1.8 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.
- 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 kerneldoc for each such case in drm_crtc.h.
- There's a new helper drm_atomic_helper_best_encoder() which could be used by all atomic drivers which don't select the encoder for a given connector at runtime. That's almost all of them, and would allow us to get rid of a lot of best_encoder boilerplate in drivers.

Contact: Daniel Vetter

16.1.9 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_free_object_unlocked 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 the following drivers still use struct_mutex: msm, omapdrm and udl.

Contact: Daniel Vetter, respective driver maintainers

16.2 Core refactorings

16.2.1 Use new IDR deletion interface to clean up drm_gem_handle_delete()

See the "This is gross" comment – apparently the IDR system now can return an error code instead of oopsing.

16.2.2 Clean up the DRM header mess

Currently the DRM subsystem has only one global header, drmP.h. This is used both for functions exported to helper libraries and drivers and functions only used internally in the drm.ko module. The goal would be to move all header declarations not needed outside of drm.ko into drivers/gpu/drm/drm_*_internal.h header files. EXPORT_SYMBOL also needs to be dropped for these functions.

This would nicely tie in with the below task to create kerneldoc after the API is cleaned up. Or with the "hide legacy cruft better" task.

Note that this is well in progress, but `drmP.h` is still huge. The updated plan is to switch to per-file driver API headers, which will also structure the kerneldoc better. This should also allow more fine-grained `#include` directives.

In the end no .c file should need to include `drmP.h` anymore.

Contact: Daniel Vetter

16.2.3 Add missing kerneldoc for exported functions

The DRM reference documentation is still lacking kerneldoc in a few areas. The task would be to clean up interfaces like moving functions around between files to better group them and improving the interfaces like dropping return values for functions that never fail. Then write kerneldoc for all exported functions and an overview section and integrate it all into the drm book.

See https://dri.freedesktop.org/docs/drm/ for what's there already.

Contact: Daniel Vetter

16.2.4 Hide legacy cruft better

Way back DRM supported only drivers which shadow-attached to PCI devices with userspace or fbdev drivers setting up outputs. Modern DRM drivers take charge of the entire device, you can spot them with the DRIVER_MODESET flag.

Unfortunately there's still large piles of legacy code around which needs to be hidden so that driver writers don't accidentally end up using it. And to prevent security issues in those legacy IOCTLs from being exploited on modern drivers. This has multiple possible subtasks:

- Extract support code for legacy features into a `drm-legacy.ko` kernel module and compile it only when one of the legacy drivers is enabled.

This is mostly done, the only thing left is to split up `drm_irq.c` into legacy cruft and the parts needed by modern KMS drivers.

Contact: Daniel Vetter

16.2.5 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 have one, but on top of that the fbcon code itself also has one. We need to make sure that they stop fighting over each another.
- `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.
- For the above locking troubles reasons it's pretty much impossible to attempt a synchronous modeset from panic handlers. The only thing we could try to achive is an atomic `set_base` of the primary plane, and hope that it shows up. Everything else probably needs to be delayed to some worker or

something else which happens later on. Otherwise it just kills the box harder, prevent the panic from going out on e.g. netconsole.

- There's also proposal for a simplied DRM console instead of the full-blown fbcon and DRM fbdev emulation. Any kind of panic handling tricks should obviously work for both console, in case we ever get kmslog merged.

Contact: Daniel Vetter

16.2.6 Clean up the debugfs support

There's a bunch of issues with it:

- The drm_info_list ->show() function doesn't even bother to cast to the drm structure for you. This is lazy.
- 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_info_list stuff is centered on drm_minor instead of drm_device. For anything we want to print drm_device (or maybe drm_file) is the right thing.
- 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/unpuplishing 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

16.3 Better Testing

16.3.1 Enable trinity for DRM

And fix up the fallout. Should be really interesting ...

16.3.2 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.

Contact: Daniel Vetter

16.3.3 Create a virtual KMS driver for testing (vkms)

With all the latest helpers it should be fairly simple to create a virtual KMS driver useful for testing, or for running X or similar on headless machines (to be able to still use the GPU). This would be similar to vgem, but aimed at the modeset side.

Once the basics are there there's tons of possibilities to extend it.

Contact: Daniel Vetter

16.4 Driver Specific

16.4.1 tinydrm

Tinydrm is the helper driver for really simple fb drivers. The goal is to make those drivers as simple as possible, so lots of room for refactoring:

- backlight helpers, probably best to put them into a new drm_backlight.c. This is because drivers/video is de-facto unmaintained. We could also move drivers/video/backlight to drivers/gpu/backlight and take it all over within drm-misc, but that's more work.
- spi helpers, probably best put into spi core/helper code. Thierry said the spi maintainer is fast&reactive, so shouldn't be a big issue.
- extract the mipi-dbi helper (well, the non-tinydrm specific parts at least) into a separate helper, like we have for mipi-dsi already. Or follow one of the ideas for having a shared dsi/dbi helper, abstracting away the transport details more.
- tinydrm_lastclose could be drm_fb_helper_lastclose. Only thing we need for that is to store the drm_fb_helper pointer somewhere in drm_device->mode_config. And then we could roll that out to all the drivers.
- tinydrm_gem_cma_prime_import_sg_table should probably go into the cma helpers, as a _vmapped variant (since not every driver needs the vmap). And tinydrm_gem_cma_free_object could the be merged into drm_gem_cma_free_object().
- tinydrm_fb_create we could move into drm_simple_pipe, only need to add the fb_create hook to drm_simple_pipe_funcs, which would again simplify a bunch of things (since it gives you a one-stop vfunc for simple drivers).
- Quick aside: The unregister devm stuff is kinda getting the lifetimes of a drm_device wrong. Doesn't matter, since everyone else gets it wrong too :-)
- With the fbdev pointer in dev->mode_config we could also make suspend/resume helpers entirely generic, at least if we add a dev->mode_config.suspend_state. We could even provide a generic pm_ops structure with those.
- also rework the drm_framebuffer_funcs->dirty hook wire-up, see above.

Contact: Noralf Trønnes, Daniel Vetter

16.5 Outside DRM

Symbols

A

B

C

D

F

G

H

M

O

P

R

S

T

V