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.

Core Driver Infrastructure

This section covers core driver infrastructure used by both the display and the GEM parts of the driver.

Runtime Power Management

The i915 driver supports dynamic enabling and disabling of entire hardware blocks at runtime. This is especially important on the display side where software is supposed to control many power gates manually on recent hardware, since on the GT side a lot of the power management is done by the hardware. But even there some manual control at the device level is required.

Since i915 supports a diverse set of platforms with a unified codebase and hardware engineers just love to shuffle functionality around between power domains there's a sizeable amount of indirection required. This file provides generic functions to the driver for grabbing and releasing references for abstract power domains. It then maps those to the actual power wells present for a given platform.

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.

Interrupt Handling

These functions provide the basic support for enabling and disabling the interrupt handling support. There's a lot more functionality in i915_irq.c and related files, but that will be described in separate chapters.

---

void **intel_irq_init**(struct drm_i915_private * ***dev_priv***)

initializes irq support

Parameters

`struct drm_i915_private * dev_priv`
i915 device instance

Description

This function initializes all the irq support including work items, timers and all the vtables. It does not setup the interrupt itself though.

---

void **intel_runtime_pm_disable_interrupts**(struct drm_i915_private * ***dev_priv***)

runtime interrupt disabling

Parameters

`struct drm_i915_private * dev_priv`
i915 device instance

Description

This function is used to disable interrupts at runtime, both in the runtime pm and the system suspend/resume code.

---

void **intel_runtime_pm_enable_interrupts**(struct drm_i915_private * ***dev_priv***)

runtime interrupt enabling

Parameters

`struct drm_i915_private * dev_priv`
i915 device instance

Description

This function is used to enable interrupts at runtime, both in the runtime pm and the system suspend/resume code.

Intel GVT-g Guest Support(vGPU)

Intel GVT-g is a graphics virtualization technology which shares the GPU among multiple virtual machines on a time-sharing basis. Each virtual machine is presented a virtual GPU (vGPU), which has equivalent features as the underlying physical GPU (pGPU), so i915 driver can run seamlessly in a virtual machine. This file provides vGPU specific optimizations when running in a virtual machine, to reduce the complexity of vGPU emulation and to improve the overall performance.

A primary function introduced here is so-called “address space ballooning” technique. Intel GVT-g partitions global graphics memory among multiple VMs, so each VM can directly access a portion of the memory without hypervisor’s intervention, e.g. filling textures or queuing commands. However with the partitioning an unmodified i915 driver would assume a smaller graphics memory starting from address ZERO, then requires vGPU emulation module to translate the graphics address between ‘guest view’ and ‘host view’, for all registers and command opcodes which contain a graphics memory address. To reduce the complexity, Intel GVT-g introduces “address space ballooning”, by telling the exact partitioning knowledge to each guest i915 driver, which then reserves and prevents non-allocated portions from allocation. Thus vGPU emulation module only needs to scan and validate graphics addresses without complexity of address translation.

void **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

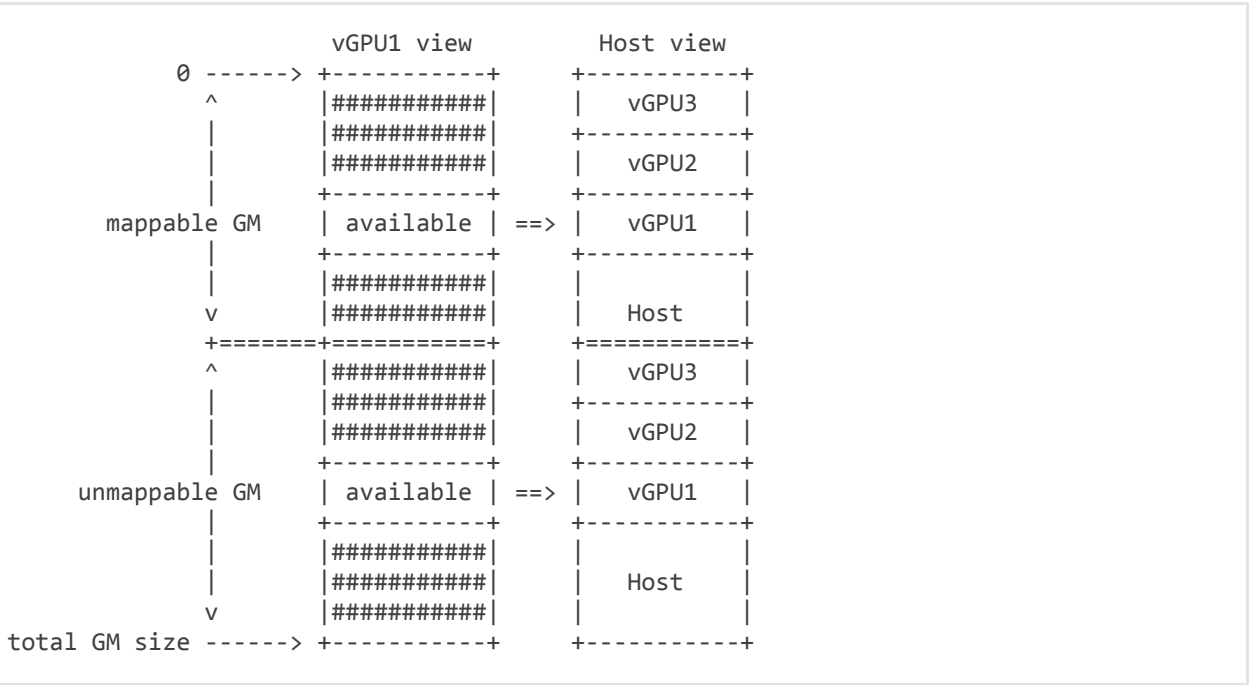
Parameters

`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.



Return

zero on success, non-zero if configuration invalid or ballooning failed

Intel GVT-g Host Support(vGPU device model)

Intel GVT-g is a graphics virtualization technology which shares the GPU among multiple virtual machines on a time-sharing basis. Each virtual machine is presented a virtual GPU (vGPU), which has equivalent features as the underlying physical GPU (pGPU), so i915 driver can run seamlessly in a virtual machine.

To virtualize GPU resources GVT-g driver depends on hypervisor technology e.g KVM/VFIO/mdev, Xen, etc. to provide resource access trapping capability and be virtualized within GVT-g device module. More architectural design doc is available on https://01.org/group/2230/documentation-list.

---

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.

Display Hardware Handling

This section covers everything related to the display hardware including the mode setting infrastructure, plane, sprite and cursor handling and display, output probing and related topics.

Mode Setting Infrastructure

The i915 driver is thus far the only DRM driver which doesn't use the common DRM helper code to implement mode setting sequences. Thus it has its own tailor-made infrastructure for executing a display configuration change.

Frontbuffer Tracking

Many features require us to track changes to the currently active frontbuffer, especially rendering targeted at the frontbuffer.

To be able to do so 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.

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 pipe ***pch_transcoder***, bool ***enable***)

set PCH fifo underrun reporting state

Parameters

`struct drm_i915_private * dev_priv`
i915 device instance

`enum pipe pch_transcoder`
the PCH transcoder (same as pipe on IVB and older)

`bool enable`
whether underruns should be reported or not

Description

This function makes us disable or enable PCH fifo underruns for a specific PCH transcoder. Notice that on some PCHs (e.g. CPT/PPT), disabling FIFO underrun reporting for one transcoder may also disable all the other PCH error 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 pipe ***pch_transcoder***)

handle PCH fifo underrun interrupt

Parameters

`struct drm_i915_private * dev_priv`
i915 device instance

`enum pipe pch_transcoder`
the PCH transcoder (same as pipe on IVB and older)

Description

This handles a PCH fifo underrun interrupt, generating an underrun warning into dmesg if underrun reporting is enabled and then disables the underrun interrupt to avoid an irq storm.

void **intel_check_cpu_fifo_underruns**(struct drm_i915_private * ***dev_priv***)

check for CPU fifo underruns immediately

Parameters

`struct drm_i915_private * dev_priv`
i915 device instance

Description

Check for CPU fifo underruns immediately. Useful on IVB/HSW where the shared error interrupt may have been disabled, and so CPU fifo underruns won't necessarily raise an interrupt, and on GMCH platforms where underruns never raise an interrupt.

void **intel_check_pch_fifo_underruns**(struct drm_i915_private * ***dev_priv***)

check for PCH fifo underruns immediately

Parameters

`struct drm_i915_private * dev_priv`
i915 device instance

Description

Check for PCH fifo underruns immediately. Useful on CPT/PPT where the shared error interrupt may have been disabled, and so PCH fifo underruns won't necessarily raise an interrupt.

Plane Configuration

This section covers plane configuration and composition with the primary plane, sprites, cursors and overlays. This includes the infrastructure to do atomic vsync'ed updates of all this state and also tightly coupled topics like watermark setup and computation, framebuffer compression and panel self refresh.

Atomic Plane Helpers

The functions here are used by the atomic plane helper functions to implement legacy plane updates (i.e., drm_plane->: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

Output Probing

This section covers output probing and related infrastructure like the hotplug interrupt storm detection and mitigation code. Note that the i915 driver still uses most of the common DRM helper code for output probing, so those sections fully apply.

Hotplug

Simply put, hotplug occurs when a display is connected to or disconnected from the system. However, there may be adapters and docking stations and Display Port short pulses and MST devices involved, complicating matters.

Hotplug in i915 is handled in many different levels of abstraction.

The platform dependent interrupt handling code in i915_irq.c enables, disables, and does preliminary handling of the interrupts. The interrupt handlers gather the hotplug detect (HPD) information from relevant registers into a platform independent mask of hotplug pins that have fired.

The platform independent interrupt handler `intel_hpd_irq_handler()` in intel_hotplug.c does hotplug irq storm detection and mitigation, and passes further processing to appropriate bottom halves (Display Port specific and regular hotplug).

The Display Port work function `i915_digport_work_func()` calls into `intel_dp_hpd_pulse()` via hooks, which handles DP short pulses and DP MST long pulses, with failures and non-MST long pulses triggering regular hotplug processing on the connector.

The regular hotplug work function `i915_hotplug_work_func()` calls connector detect hooks, and, if connector status changes, triggers sending of hotplug uevent to userspace via `drm_kms_helper_hotplug_event()`.

Finally, the userspace is responsible for triggering a modeset upon receiving the hotplug uevent, disabling or enabling the crtc as needed.

The hotplug interrupt storm detection and mitigation code keeps track of the number of interrupts per hotplug pin per a period of time, and if the number of interrupts exceeds a certain threshold, the interrupt is disabled for a while before being re-enabled. The intention is to mitigate issues raising from broken hardware triggering massive amounts of interrupts and grinding the system to a halt.

Current implementation expects that hotplug interrupt storm will not be seen when display port sink is connected, hence on platforms whose DP callback is handled by i915_digport_work_func reenabling of hpd is not performed (it was never expected to be disabled in the first place ;)) this is specific to DP sinks handled by this routine and any other display such as HDMI or DVI enabled on the same port will have proper logic since it will use i915_hotplug_work_func where this logic is handled.

enum port **intel_hpd_pin_to_port**(enum hpd_pin *pin*)

return port hard associated with certain pin.

Parameters

`enum hpd_pin pin`
the hpd pin to get associated port

Description

Return port that is associatade with **pin** and PORT_NONE if no port is hard associated with that **pin**.

enum hpd_pin **intel_hpd_pin**(enum port *port*)

return pin hard associated with certain port.

Parameters

`enum port port`
the hpd port to get associated pin

Description

Return pin that is associatade with **port** and HDP_NONE if no pin is hard associated with that **port**.

---

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.

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

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.

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.

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.

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).

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
-----------------
```

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.

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)

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

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.

Memory Management and Command Submission

This sections covers all things related to the GEM implementation in the i915 driver.

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

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

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.

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.

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.

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.

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.

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 long * ***nr_scanned***, 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 long * nr_scanned`
optional output for number of pages scanned (incremental)

`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.

GuC

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

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

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.

Tracing

This sections covers all things related to the tracepoints implemented in the i915 driver.

i915_ppgtt_create and i915_ppgtt_release

With full ppgtt enabled each process using drm will allocate at least one translation table. With these traces it is possible to keep track of the allocation and of the lifetime of the tables; this can be used during testing/debug to verify that we are not leaking ppgtts. These traces identify the ppgtt through the vm pointer, which is also printed by the i915_vma_bind and i915_vma_unbind tracepoints.

i915_context_create and i915_context_free

These tracepoints are used to track creation and deletion of contexts. If full ppgtt is enabled, they also print the address of the vm assigned to the context.

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.

Perf

Overview

Gen graphics supports a large number of performance counters that can help driver and application developers understand and optimize their use of the GPU.

This i915 perf interface enables userspace to configure and open a file descriptor representing a stream of GPU metrics which can then be `read()` as a stream of sample records.

The interface is particularly suited to exposing buffered metrics that are captured by DMA from the GPU, unsynchronized with and unrelated to the CPU.

Streams representing a single context are accessible to applications with a corresponding drm file descriptor, such that OpenGL can use the interface without special privileges. Access to system-wide metrics requires root privileges by default, unless changed via the dev.i915.perf_event_paranoid sysctl option.

Comparison with Core Perf

The interface was initially inspired by the core Perf infrastructure but some notable differences are:

i915 perf file descriptors represent a “stream” instead of an “event”; where a perf event primarily corresponds to a single 64bit value, while a stream might sample sets of tightly-coupled counters, depending on the configuration. For example the Gen OA unit isn’t designed to support orthogonal configurations of individual counters; it’s configured for a set of related counters. Samples for an i915 perf stream capturing OA metrics will include a set of counter values packed in a compact HW specific format. The OA unit supports a number of different packing formats which can be selected by the user opening the stream. Perf has support for grouping events, but each event in the group is configured, validated and authenticated individually with separate system calls.

i915 perf stream configurations are provided as an array of u64 (key,value) pairs, instead of a fixed struct with multiple miscellaneous config members, interleaved with event-type specific members.

i915 perf doesn’t support exposing metrics via an mmap’d circular buffer. The supported metrics are being written to memory by the GPU unsynchronized with the CPU, using HW specific packing formats for counter sets. Sometimes the constraints on HW configuration require reports to be filtered before it would be acceptable to expose them to unprivileged applications - to hide the metrics of other processes/contexts. For these use cases a `read()` based interface is a good fit, and provides an opportunity to filter data as it gets copied from the GPU mapped buffers to userspace buffers.

Issues hit with first prototype based on Core Perf

The first prototype of this driver was based on the core perf infrastructure, and while we did make that mostly work, with some changes to perf, we found we were breaking or working around too many assumptions baked into perf’s currently cpu centric design.

In the end we didn't see a clear benefit to making perf's implementation and interface more complex by changing design assumptions while we knew we still wouldn't be able to use any existing perf based userspace tools.

Also considering the Gen specific nature of the Observability hardware and how userspace will sometimes need to combine i915 perf OA metrics with side-band OA data captured via MI_REPORT_PERF_COUNT commands; we're expecting the interface to be used by a platform specific userspace such as OpenGL or tools. This is to say; we aren't inherently missing out on having a standard vendor/architecture agnostic interface by not using perf.

For posterity, in case we might re-visit trying to adapt core perf to be better suited to exposing i915 metrics these were the main pain points we hit:

- The perf based OA PMU driver broke some significant design assumptions:

  Existing perf pmus are used for profiling work on a cpu and we were introducing the idea of _IS_DEVICE pmus with different security implications, the need to fake cpu-related data (such as user/kernel registers) to fit with perf's current design, and adding _DEVICE records as a way to forward device-specific status records.

  The OA unit writes reports of counters into a circular buffer, without involvement from the CPU, making our PMU driver the first of a kind.

  Given the way we were periodically forward data from the GPU-mapped, OA buffer to perf's buffer, those bursts of sample writes looked to perf like we were sampling too fast and so we had to subvert its throttling checks.

  Perf supports groups of counters and allows those to be read via transactions internally but transactions currently seem designed to be explicitly initiated from the cpu (say in response to a userspace `read()`) and while we could pull a report out of the OA buffer we can't trigger a report from the cpu on demand.

  Related to being report based; the OA counters are configured in HW as a set while perf generally expects counter configurations to be orthogonal. Although counters can be associated with a group leader as they are opened, there's no clear precedent for being able to provide group-wide configuration attributes (for example we want to let userspace choose the OA unit report format used to capture all counters in a set, or specify a GPU context to filter metrics on). We

avoided using perf's grouping feature and forwarded OA reports to userspace via perf's 'raw' sample field. This suited our userspace well considering how coupled the counters are when dealing with normalizing. It would be inconvenient to split counters up into separate events, only to require userspace to recombine them. For Mesa it's also convenient to be forwarded raw, periodic reports for combining with the side-band raw reports it captures using MI_REPORT_PERF_COUNT commands.

- As a side note on perf's grouping feature; there was also some concern that using PERF_FORMAT_GROUP as a way to pack together counter values would quite drastically inflate our sample sizes, which would likely lower the effective sampling resolutions we could use when the available memory bandwidth is limited.

  With the OA unit's report formats, counters are packed together as 32 or 40bit values, with the largest report size being 256 bytes.

  PERF_FORMAT_GROUP values are 64bit, but there doesn't appear to be a documented ordering to the values, implying PERF_FORMAT_ID must also be used to add a 64bit ID before each value; giving 16 bytes per counter.

Related to counter orthogonality; we can't time share the OA unit, while event scheduling is a central design idea within perf for allowing userspace to open + enable more events than can be configured in HW at any one time. The OA unit is not designed to allow re-configuration while in use. We can't reconfigure the OA unit without losing internal OA unit state which we can't access explicitly to save and restore. Reconfiguring the OA unit is also relatively slow, involving ~100 register writes. From userspace Mesa also depends on a stable OA configuration when emitting MI_REPORT_PERF_COUNT commands and importantly the OA unit can't be disabled while there are outstanding MI_RPC commands lest we hang the command streamer.

The contents of sample records aren't extensible by device drivers (i.e. the sample_type bits). As an example; Sourab Gupta had been looking to attach GPU timestamps to our OA samples. We were shoehorning OA reports into sample records by using the 'raw' field, but it's tricky to pack more than one thing into this field because events/core.c currently only lets a pmu give a single raw data pointer plus len which will be copied into the ring buffer. To include more than the OA report we'd have to copy the report into an intermediate larger buffer. I'd been considering allowing a vector of data+len values to be specified for copying the raw data, but it felt like a kludge to being using the raw field for this purpose.

- It felt like our perf based PMU was making some technical compromises just for the sake of using perf:

  `perf_event_open()` requires events to either relate to a pid or a specific cpu core, while our device pmu related to neither. Events opened with a pid will be automatically enabled/disabled according to the scheduling of that process - so not appropriate for us. When an event is related to a cpu id, perf ensures pmu methods will be invoked via an inter process interrupt on that core. To avoid invasive changes our userspace opened OA perf events for a specific cpu. This was workable but it meant the majority of the OA driver ran in atomic context, including all OA report forwarding, which wasn't really necessary in our case and seems to make our locking requirements somewhat complex as we handled the interaction with the rest of the i915 driver.

i915 Driver Entry Points

This section covers the 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.

---

int **i915_perf_add_config_ioctl**(struct drm_device * *dev*, void * *data*, struct drm_file * *file*)

DRM `ioctl()` for userspace to add a new OA config

Parameters

`struct drm_device * dev`
drm device

`void * data`
ioctl data (pointer to struct drm_i915_perf_oa_config) copied from userspace (unvalidated)

`struct drm_file * file`
drm file

Description

Validates the submitted OA register to be saved into a new OA config that can then be used for programming the OA unit and its NOA network.

Return

A new allocated config number to be used with the perf open ioctl or a negative error code on failure.

int **i915_perf_remove_config_ioctl**(struct drm_device * ***dev***, void * ***data***, struct drm_file * ***file***)

DRM `ioctl()` for userspace to remove an OA config

Parameters

`struct drm_device * dev`
drm device

`void * data`
ioctl data (pointer to u64 integer) copied from userspace

`struct drm_file * file`
drm file

Description

Configs can be removed while being used, the will stop appearing in sysfs and their content will be freed when the stream using the config is closed.

Return

0 on success or a negative error code on failure.

i915 Perf Stream

This section covers the stream-semantics-agnostic structures and functions for representing an i915 perf stream FD and associated file operations.

struct **i915_perf_stream**

state for a single open stream FD

Definition

```
struct i915_perf_stream {
  struct 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;
  struct i915_oa_config * oa_config;
};
```

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.

`oa_config`
The OA configuration used by the 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

i915 Perf Observation Architecture Stream

struct i915_oa_ops

Gen specific implementation of an OA unit stream

Definition

```
struct i915_oa_ops {
  bool (* is_valid_b_counter_reg) (struct drm_i915_private *dev_priv, u32 addr);
  bool (* is_valid_mux_reg) (struct drm_i915_private *dev_priv, u32 addr);
  bool (* is_valid_flex_reg) (struct drm_i915_private *dev_priv, u32 addr);
  void (* init_oa_buffer) (struct drm_i915_private *dev_priv);
  int (* enable_metric_set) (struct drm_i915_private *dev_priv, const struct
i915_oa_config *oa_config);
  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

is_valid_b_counter_reg
Validates register's address for programming boolean counters for a particular platform.

is_valid_mux_reg
Validates register's address for programming mux for a particular platform.

is_valid_flex_reg
Validates register's address for programming flex EU filtering for a particular platform.

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)

`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.

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()`.

---

int **i915_perf_add_config_ioctl**(struct drm_device * *dev*, void * *data*, struct drm_file * *file*)

DRM `ioctl()` for userspace to add a new OA config

Parameters

`struct drm_device * dev`
drm device

`void * data`
ioctl data (pointer to struct drm_i915_perf_oa_config) copied from userspace (unvalidated)

`struct drm_file * file`
drm file

Description

Validates the submitted OA register to be saved into a new OA config that can then be used for programming the OA unit and its NOA network.

Return

A new allocated config number to be used with the perf open ioctl or a negative error code on failure.

---

int **i915_perf_remove_config_ioctl**(struct drm_device * *dev*, void * *data*, struct drm_file * *file*)

DRM `ioctl()` for userspace to remove an OA config

Parameters

`struct drm_device * dev`
drm device

`void * data`
ioctl data (pointer to u64 integer) copied from userspace

`struct drm_file * file`
drm file

Description

Configs can be removed while being used, the will stop appearing in sysfs and their content will be freed when the stream using the config is closed.

Return

0 on success or a negative error code on failure.

---

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

Style

The drm/i915 driver codebase has some style rules in addition to (and, in some cases, deviating from) the kernel coding style.

Register macro definition style

The style guide for `i915_reg.h`.

Follow the style described here for new macros, and while changing existing macros. Do **not** mass change existing definitions just to update the style.

Layout

Keep helper macros near the top. For example, `_PIPE()` and friends.

Prefix macros that generally should not be used outside of this file with underscore '_'. For example, `_PIPE()` and friends, single instances of registers that are defined solely for the use by function-like macros.

Avoid using the underscore prefixed macros outside of this file. There are exceptions, but keep them to a minimum.

There are two basic types of register definitions: Single registers and register groups. Register groups are registers which have two or more instances, for example one per pipe, port, transcoder, etc. Register groups should be defined using function-like macros.

For single registers, define the register offset first, followed by register contents.

For register groups, define the register instance offsets first, prefixed with underscore, followed by a function-like macro choosing the right instance based on the parameter, followed by register contents.

Define the register contents (i.e. bit and bit field macros) from most significant to least significant bit. Indent the register content macros using two extra spaces between `#define` and the macro name.

For bit fields, define a `_MASK` and a `_SHIFT` macro. Define bit field contents so that they are already shifted in place, and can be directly OR'd. For convenience, function-like macros may be used to define bit fields, but do note that the macros may be needed to read as well as write the register contents.

Define bits using `(1 << N)` instead of `BIT(N)`. We may change this in the future, but this is the prevailing style. Do **not** add `_BIT` suffix to the name.

Group the register and its contents together without blank lines, separate from other registers and their contents with one blank line.

Indent macro values from macro names using TABs. Align values vertically. Use braces in macro values as needed to avoid unintended precedence after macro substitution. Use spaces in macro values according to kernel coding style. Use lower case in hexadecimal values.

Naming

Try to name registers according to the specs. If the register name changes in the specs from platform to another, stick to the original name.

Try to re-use existing register macro definitions. Only add new macros for new register offsets, or when the register contents have changed enough to warrant a full redefinition.

When a register macro changes for a new platform, prefix the new macro using the platform acronym or generation. For example, `SKL_` or `GEN8_`. The prefix signifies the start platform/generation using the register.

When a bit (field) macro changes or gets added for a new platform, while retaining the existing register macro, add a platform acronym or generation suffix to the name. For example, `_SKL` or `_GEN8`.

Examples

(Note that the values in the example are indented using spaces instead of TABs to avoid misalignment in generated documentation. Use TABs in the definitions.):

```
#define _FOO_A                  0xf000
#define _FOO_B                  0xf001
#define FOO(pipe)               _MMIO_PIPE(pipe, _FOO_A, _FOO_B)
#define   FOO_ENABLE            (1 << 31)
#define   FOO_MODE_MASK         (0xf << 16)
#define   FOO_MODE_SHIFT        16
#define   FOO_MODE_BAR          (0 << 16)
#define   FOO_MODE_BAZ          (1 << 16)
#define   FOO_MODE_QUX_SNB      (2 << 16)

#define BAR                     _MMIO(0xb000)
#define GEN8_BAR                _MMIO(0xb888)
```