

Nova — Building an NVIDIA GPU Driver in Rust Upstream

Linux Plumbers Conference 2026

Danilo Krummrich
Red Hat

John Hubbard
NVIDIA

**LINUX
PLUMBERS CONFERENCE**

PRAGUE, CZECHIA
OCT. 5-7, 2026



Intro to Nova

An upstream Linux kernel driver for NVIDIA GPUs, in Rust

A combined graphics and compute driver, for bare metal and vGPU.

Entirely upstream-based and community-based

Design, coding and reviews all happen publicly.

Long term, a replacement for Open RM, but not a "drop-in replacement"

New UAPI, so lower level libraries need a new adapter layer, to talk to Nova. Top level Vulkan apps are likely unchanged.

Support from the Rust for Linux community has been amazing!

In-depth reviews, and supporting patchsets from the Rust for Linux developers has made a huge difference.

Nova architecture

Vulkan Compute · CUDA · Graphics
OpenCL · tools

/dev/dri/card[0..N] · fwctl · sysfs

nova-drm.ko

graphics + compute, on DRM (TTM, GEM, ...)

aux bus

QEMU · VMware · VFIO management
VMs

KVM · VFIO

nvidia-vfio-vgpu.ko

small host/hypervisor driver
== small attack surface

nova-core.ko

GSP-RM · GSP plugin for vGPU · GPU BARo registers

Turing · Ampere · Ada · Hopper · Blackwell · Future Chips

Nova: status

nova-core

MAINLINE

GSP (GPU firmware) boots and unloads cleanly on Turing through Blackwell (Hopper remains untested)

- Uses r570.144 as a temporary firmware set, to be deleted

Basic GPU interrupt support

Basic BAR1 support

POSTED

Support for r615+ firmware, GMC

nova-drm

MAINLINE

Auxiliary driver on top of nova-core

- Pure Rust to Rust with nova-core: no C API in between

QUEUED

GPU info ioctl (drm-rust-next)

- `drm-test`¹: Rust app that queries and prints GPU info
- `deviceQuery`² (from CUDA samples on GitHub): partially working

vGPU / VFIO / virtualization

PoC / Posted

Zhi Wang's public PoC tree runs Linux and Windows guests in VMs

- All required patches for this are posted publicly

¹ gitlab.freedesktop.org/dakr/drm-test

² github.com/NVIDIA/cuda-samples/tree/master/cpp/1_Uutilities/deviceQuery

Nova: what's next

Virtualization

BASIC

- Linux and Windows guests in VMs

ADVANCED

- MIG
- VM live migration
- Stable GSP ABI for paravirt calls
- Per-VM runlists

Bare metal

BASIC

- Vulkan Compute and CUDA apps
- Graphics
- Display
- Assorted supporting features

NEEDED TO LAUNCH A BASIC CUDA OR GRAPHICS APP

1. Finish the basic vidmem (VRAM) allocator
2. Channel management
3. GR golden context

Vulkan Compute on Nova

Pinned vidmem and user mode command submission

- **Quick to develop**

No vidmem eviction or paging to get right first.

- **Best performance**

Work submission doesn't go through the kernel.

- **PyTorch ready**

Matches how major AI workloads already work.

```
1 /* GLSL shader excerpt (GPU code): */
2 void main() { val = val * val; }
3
4 /* Host (CPU) code: */
5 *val = 25; /* mapped, host-coherent buffer */
6
7 vkBeginCommandBuffer(cmd, &begin);
8 vkCmdBindPipeline(cmd,
9     VK_PIPELINE_BIND_POINT_COMPUTE, pipeline);
10
11 vkCmdBindDescriptorSets(cmd,
12     VK_PIPELINE_BIND_POINT_COMPUTE,
13     layout, 0, 1, &set, 0, NULL);
14
15 vkCmdDispatch(cmd, 1, 1, 1);
16 vkCmdPipelineBarrier(...);
17 vkEndCommandBuffer(cmd);
18
19 /* user mode submission: no ioctl */
20 vkQueueSubmit(queue, 1, &submit, fence);
21 vkWaitForFences(dev, 1, &fence,
22     VK_TRUE, UINT64_MAX);
23 printf("Square(25) = %d\n", *val);
```

Paving the Way: Stable GMC APIs

NVIDIA's GSP-RM firmware team is providing stable GSP APIs before each Nova feature that depends on them.

These are called "GMC (GPU Management Controller) APIs".

Why GSP-RM APIs must be stable

- Nova loads GSP firmware from linux-firmware, which is released independently of the kernel.
- Updating either linux-firmware or kernel must not break booting, or loading Nova.

Therefore:

- Nova restricts itself to using GMC APIs, as opposed to the older "RPC RM APIs".

This rule was relaxed during bring-up, in order to temporarily use the pre-existing, non-stable r570.144 firmware.

Upcoming drivers (after r615) will have a stable GSP-RM ABI (eventually / soon).

HAL architecture

one driver, six architectures, twenty-two chipsets...so far

22

chipsets
supported

6

GPU
architectures

6

HAL trait
families

FbHal · GpuHal · GspHal
FalconHal · FspHal · VgpuHal
in nova-core today

Rust is far more expressive than C: Nova HALs are written directly in pure Rust.

- Most operations are shared across generations; a minority differ per chip.
- New chips reuse older implementations: GB20x uses Turing's FRTS size and Ampere's VRAM sizing.
- The Rust compiler enforces implementing the HALs for each new chip.

Nova HALs in Rust

drivers/gpu/nova-core/fb/hal.rs, v7.3-rc4

```
1 pub(crate) trait FbHal {
2     fn vidmem_size(&self, bar: Bar0<'>) -> u64;
3     fn supports_display(&self, bar: Bar0<'>) -> bool;
4     fn frts_size(&self) -> u64;
5     // ... 4 more
6 }
7
8 fn fb_hal(chipset: Chipset) -> &'static dyn FbHal {
9     match chipset.arch() {
10         Architecture::Turing => tu102::TU102_HAL,
11         Architecture::Hopper  => gh100::GH100_HAL,
12         Architecture::BlackwellGB20x =>
13             gb202::GB202_HAL,
14         // ...
15     }
16 }
```

Rust traits instead of Perl, NVOC, function pointers

Rust enforces the HAL

A chip either implements every FbHal method, or it doesn't compile.

Easily readable on-screen

No need to deal with generated code, in order to see how things are connected.

Nova: FB HAL for Blackwell

fb/hal/gb202.rs: Blackwell GB20x excerpt

```
1 impl FbHal for Gb202 {
2     fn read_sysmem_flush_page(
3         &self, bar: Bar0<'_>,
4     ) -> u64 {
5         // New for GB20x
6         read_sysmem_flush_page_gb202(bar)
7     }
8     fn supports_display(&self, bar: Bar0<'_>) -> bool {
9         // Ampere
10        super::ga100::display_enabled_ga100(bar)
11    }
12    fn vidmem_size(&self, bar: Bar0<'_>) -> u64 {
13        // Ampere
14        super::ga102::vidmem_size_ga102(bar)
15    }
16    fn frts_size(&self) -> u64 {
17        // Turing
18        super::tu102::frts_size_tu102()
19    }
20 }
```

```
1 struct Gb202;
2 const GB202: Gb202 = Gb202;
3 pub(super) const GB202_HAL: &dyn FbHal = &GB202;
```

Nouveau HALS in C, for reference

```
1 /* nouveau: nvkm/subdev/fb/priv.h */
2 struct nvkm_fb_func {
3     void (*init)(struct nvkm_fb *);
4     struct {
5         int regions;
6         void (*prog)(struct nvkm_fb *, int i,
7                     struct nvkm_fb_tile *);
8     } tile;
9     /* ... */
10 };
11
12 /* nvkm/subdev/fb/base.c */
13 if (fb->func->tile.prog)
14     fb->func->tile.prog(fb, region, tile);
15
16 for (i = 0; i < fb->tile.regions; i++)
17     fb->func->tile.prog(fb, i, &fb->tile.region[i]);
```

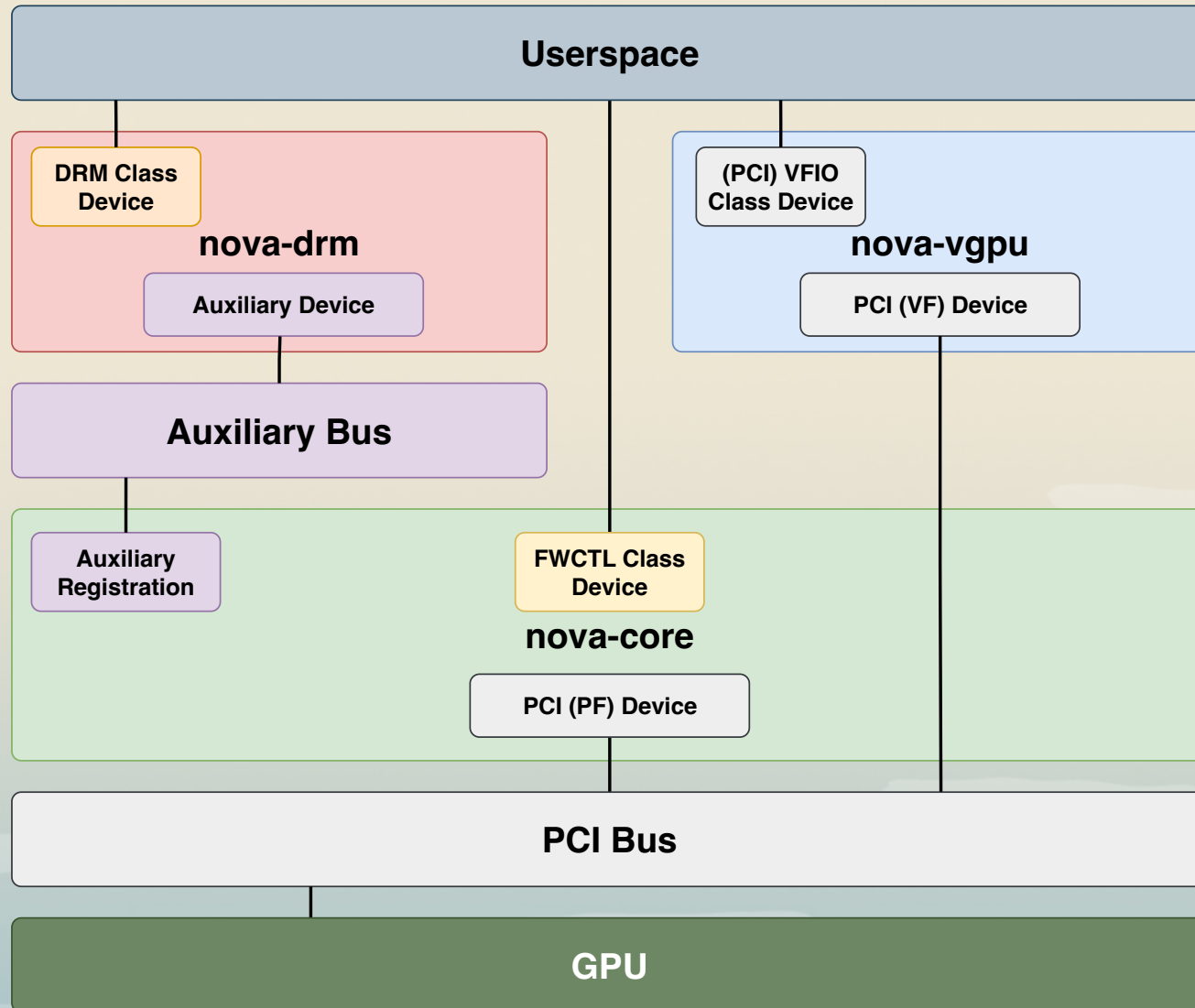
Lifetimes, Ownership, the Driver Model, DRM and Rust

LINUX
PLUMBERS CONFERENCE

PRAGUE, CZECHIA
OCT. 5-7, 2026



Nova — Architecture



Devices, Buses, and Drivers

- **Bus device:** represents a physical device on a bus (e.g. PCI)
- **Class device:** represents a certain class of devices to userspace (e.g. DRM)
- **Driver:** binds to matching devices; `probe()` creates per-device state
- **Bus:** matches devices to drivers and mediates callbacks

⇒ A driver instance operates a bus device for the duration it is bound to the driver; devices resources acquired by the must not outlive this scope.



Device Resource Cleanup

Manual (Procedural)

- Callers must ensure they call the correct cleanup function
- Must perform cleanup in the correct order
- Must clean up at the correct scope boundary (i.e. before driver unbind)

Devres (Managed)

- Revokes the resource at driver unbind
- Mostly ensures correct ordering
- Leaves stale pointers behind





Driver Model in Rust

„Enforcing Device Driver Lifecycle Rules at Compile Time“

Danilo Krummrich, Kernel Recipes 2026 [1]

[1] „Enforcing Device Driver Lifecycle Rules at Compile Time“ — kernel-recipes.org



X
S
E
A
6

Device Resources — DMA Coherent

```
1 pub struct Coherent<'a, T: KnownSize + ?Sized> {
2     dev: &'a Device<Bound>,
3     dma_addr: DmaAddress,
4     cpu_addr: NonNull<T>,
5     dma_attrs: Attrs,
6 }
7
8 impl<'a, T: AsBytes + FromBytes> Coherent<'a, T> {
9     pub fn zeroed_with_attrs(
10         dev: &'a Device<Bound>,
11         gfp_flags: alloc::Flags,
12         dma_attrs: Attrs,
13     ) -> Result<Coherent<'a, T>>;
14 }
15
16 impl<T: KnownSize + ?Sized> Drop for Coherent<'_, T> {
17     fn drop(&mut self) {
18         unsafe {
19             bindings::dma_free_attrs(
20                 self.dev.as_raw(),
21                 T::size(self.cpu_addr.as_ptr());
22                 self.cpu_addr.as_ptr().cast(),
23                 self.dma_addr,
24                 self.dma_attrs.as_raw().
```

Device Resources — DMA Coherent

```
1 pub struct Coherent<'a, T: KnownSize + ?Sized> {
2     dev: &'a Device<Bound>,
3     dma_addr: DmaAddress,
4     cpu_addr: NonNull<T>,
5     dma_attrs: Attrs,
6 }
7
8 impl<'a, T: AsBytes + FromBytes> Coherent<'a, T> {
9     pub fn zeroed_with_attrs(
10         dev: &'a Device<Bound>,
11         gfp_flags: alloc::Flags,
12         dma_attrs: Attrs,
13     ) -> Result<Coherent<'a, T>>;
14 }
15
16 impl<T: KnownSize + ?Sized> Drop for Coherent<'_, T> {
17     fn drop(&mut self) {
18         unsafe {
19             bindings::dma_free_attrs(
20                 self.dev.as_raw(),
21                 T::size(self.cpu_addr.as_ptr());
22                 self.cpu_addr.as_ptr().cast(),
23                 self.dma_addr,
24                 self.dma_attrs.as_raw());
```

Device Resources — DMA Coherent

```
5     dma_attrs: Attrs,  
6 }  
7  
8 impl<'a, T: AsBytes + FromBytes> Coherent<'a, T> {  
9     pub fn zeroed_with_attrs(  
10         dev: &'a Device<Bound>,  
11         gfp_flags: alloc::Flags,  
12         dma_attrs: Attrs,  
13     ) -> Result<Coherent<'a, T>>;  
14 }  
15  
16 impl<T: KnownSize + ?Sized> Drop for Coherent<'_, T> {  
17     fn drop(&mut self) {  
18         unsafe {  
19             bindings::dma_free_attrs(  
20                 self.dev.as_raw(),  
21                 T::size(self.cpu_addr.as_ptr());  
22                 self.cpu_addr.as_ptr().cast(),  
23                 self.dma_addr,  
24                 self.dma_attrs.as_raw(),  
25             )  
26         }  
27     }  
28 }
```

Device Resources — IRQ Registration

```
1 pub struct IrqRequest<'a> {
2     irq: u32,
3     _dev: PhantomData<&'a Device<Bound>>,
4 }
5
6 pub struct Registration<'a, T: Handler> {
7     request: IrqRequest<'a>,
8
9     #[pin]
10    handler: T,
11
12    /// Pinned because we need address stability so that we can pass a pointer
13    /// to the callback.
14    #[pin]
15    _pin: PhantomPinned,
16 }
17
18 impl<'a, T: Handler> Registration<'a, T> {
19     /// Registers the IRQ handler with the system for the given [`IrqRequest`].
20     pub fn new(
21         request: IrqRequest<'a>,
22         flags: Flags,
23         name: &'static CStr,
24         handler: impl PinInit<T, Error> + 'a,
```

Device Resources — IRQ Registration

```
1 pub struct IrqRequest<'a> {
2     irq: u32,
3     _dev: PhantomData<&'a Device<Bound>>,
4 }
5
6 pub struct Registration<'a, T: Handler> {
7     request: IrqRequest<'a>,
8
9     #[pin]
10    handler: T,
11
12    /// Pinned because we need address stability so that we can pass a pointer
13    /// to the callback.
14    #[pin]
15    _pin: PhantomPinned,
16 }
17
18 impl<'a, T: Handler> Registration<'a, T> {
19     /// Registers the IRQ handler with the system for the given [`IrqRequest`].
20     pub fn new(
21         request: IrqRequest<'a>,
22         flags: Flags,
23         name: &'static CStr,
24         handler: impl PinInit<T, Error> + 'a,
```

Device Resources — IRQ Registration

```
1 pub struct IrqRequest<'a> {
2     irq: u32,
3     _dev: PhantomData<&'a Device<Bound>>,
4 }
5
6 pub struct Registration<'a, T: Handler> {
7     request: IrqRequest<'a>,
8
9     #[pin]
10    handler: T,
11
12    /// Pinned because we need address stability so that we can pass a pointer
13    /// to the callback.
14    #[pin]
15    _pin: PhantomPinned,
16 }
17
18 impl<'a, T: Handler> Registration<'a, T> {
19     /// Registers the IRQ handler with the system for the given [`IrqRequest`].
20     pub fn new(
21         request: IrqRequest<'a>,
22         flags: Flags,
23         name: &'static CStr,
24         handler: impl PinInit<T, Error> + 'a,
```

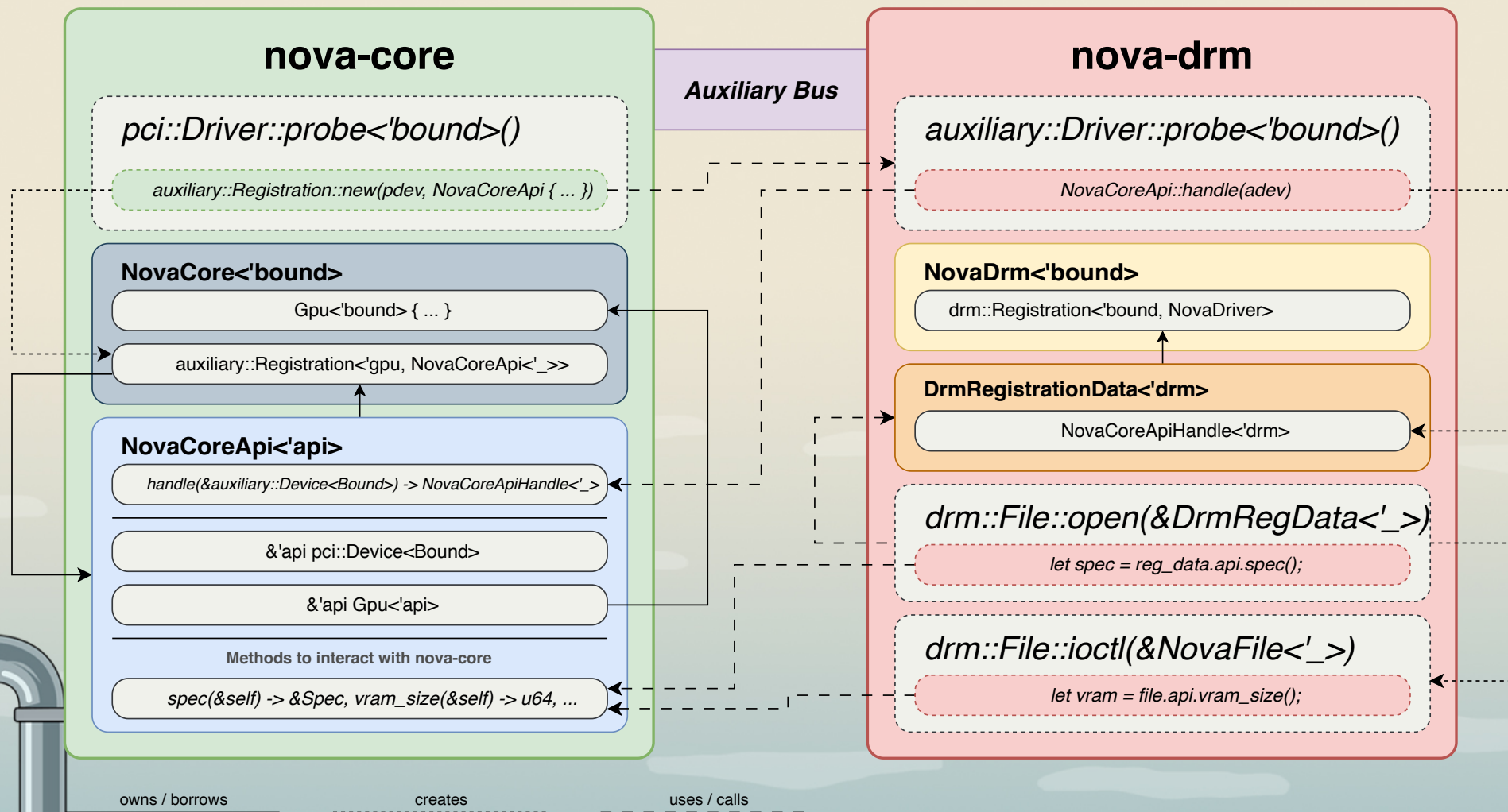
Device Resources — IRQ Registration

```
5
6 pub struct Registration<'a, T: Handler> {
7     request: IrqRequest<'a>,
8
9     #[pin]
10    handler: T,
11
12    /// Pinned because we need address stability so that we can pass a pointer
13    /// to the callback.
14    #[pin]
15    _pin: PhantomPinned,
16 }
17
18 impl<'a, T: Handler> Registration<'a, T> {
19     /// Registers the IRQ handler with the system for the given [`IrqRequest`].
20     pub fn new(
21         request: IrqRequest<'a>,
22         flags: Flags,
23         name: &'static CStr,
24         handler: impl PinInit<T, Error> + 'a,
25     ) -> impl PinInit<Self, Error> + 'a
26     where
27         T: 'a;
28 }
```

Bus Device Private Data — Rust

```
1 #[pin_data(PinnedDrop)]
2 struct DriverData<'bound> {
3     pdev: &'bound pci::Device<Bound>,
4     _enable: pci::DeviceEnableGuard<'bound>,
5     dma: Coherent<'bound, u64>,
6     #[pin]
7     irq_handler: irq::Registration<'irq_vec, IrqHandler<'irq_vec>>,
8     irq_vec: pci::IrqVectorRegistration<'bound>,
9     bar: pci::Bar<'bound>,
10 }
```

Lifecycle Across Driver Boundaries



What is RegistrationData?

```
1 void *__drm_dev_alloc(struct device *parent,  
2                       const struct drm_driver *driver,  
3                       size_t size, size_t offset);
```

```
1 impl<'a, T: Driver> Registration<'a, T> {  
2     pub fn new<E>(  
3         dev: &'a Device<Bound>,  
4         drm: drm::UnregisteredDevice<T>,  
5         data: impl PinInit<T::RegistrationData<'a>, E>  
6     ) -> impl PinInit<Self, Error>  
7     where  
8         Error: From<E>;  
9 }
```

⇒ The driver's `drm::Device` private data,
which is valid for the duration of the `drm::Device` being registered.

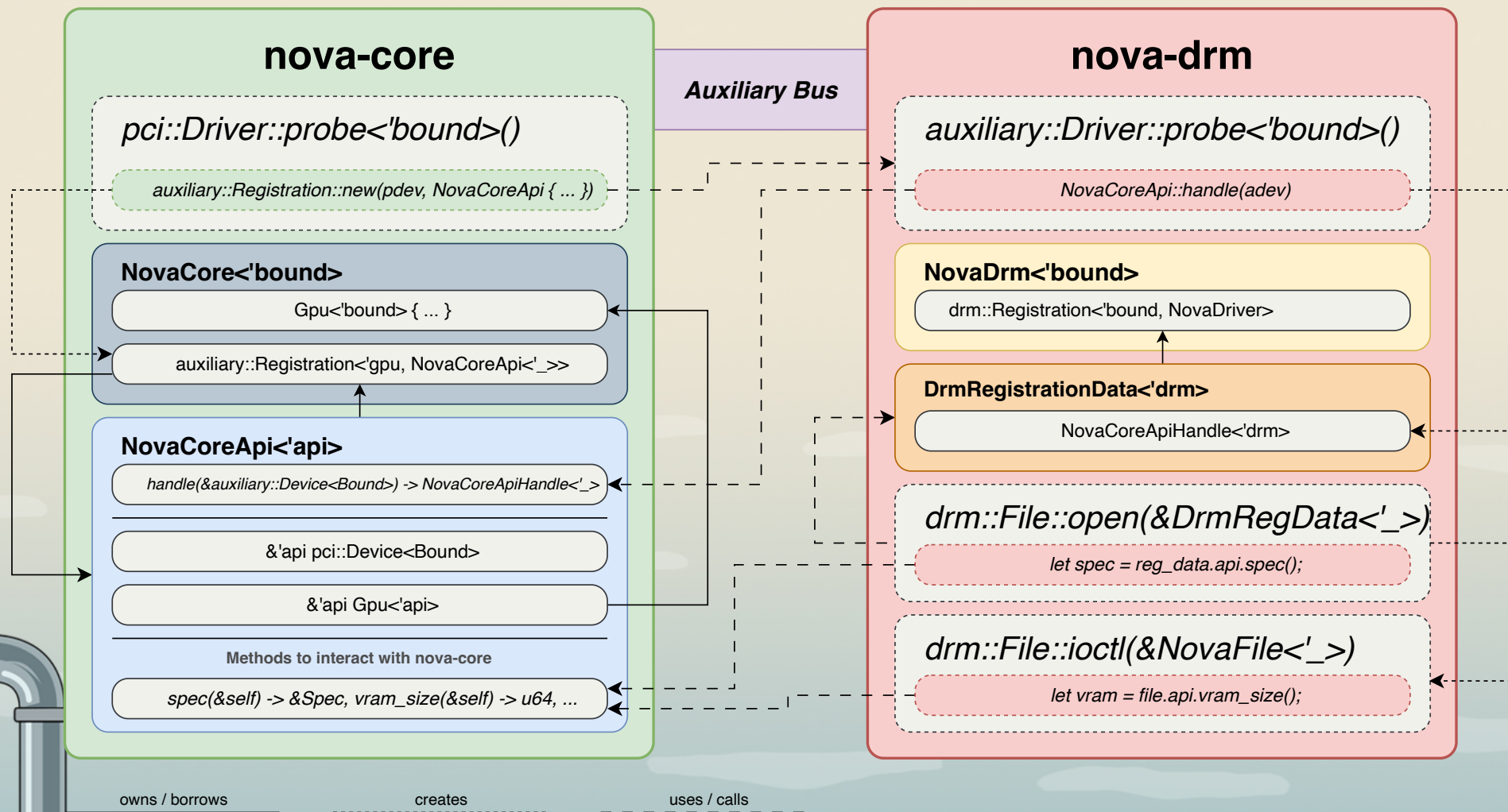
What is RegistrationData?

```
1 void *__drm_dev_alloc(struct device *parent,  
2                       const struct drm_driver *driver,  
3                       size_t size, size_t offset);
```

```
1 impl<'a, T: Driver> Registration<'a, T> {  
2     pub fn new<E>(  
3         dev: &'a Device<Bound>,  
4         drm: drm::UnregisteredDevice<T>,  
5         data: impl PinInit<T::RegistrationData<'a>, E>  
6     ) -> impl PinInit<Self, Error>  
7     where  
8         Error: From<E>;  
9 }
```

⇒ The driver's `drm::Device` private data,
which is valid for the duration of the `drm::Device` being registered.

Lifecycle Across Driver Boundaries





Why the "Wild West"?

- DRM midlayer code made the subsystem inflexible
- Midlayer code was replaced with independent helpers

⇒ *Drivers have a lot of flexibility and arrive at different solutions.*



DRM Unplug

```
1 static int panthor_ioctl_vm_create(struct drm_device *drm, void *data,  
2                                   struct drm_file *file)  
3 {  
4     int cookie, ret;  
5  
6     if (!drm_dev_enter(drm, &cookie))  
7         return -ENODEV;  
8  
9     ret = panthor_vm_pool_create_vm(...);  
10  
11     drm_dev_exit(cookie);  
12  
13     return ret;  
14 }
```

⇒ *Synchronized against `drm_dev_unplug()`.*

Xe — Lifecycle and Ownership

```
1 int xe_mmio_probe_early(struct xe_device *xe)
2 {
3     struct pci_dev *pdev = to_pci_dev(xe->drm.dev);
4
5     xe->mmio.regs = pcim_iomap(pdev, GTTMMADR_BAR, 0);
6     if (!xe->mmio.regs) {
7         xe_err(xe, "Failed to map GTTMMADR_BAR\n");
8         return -EIO;
9     }
10
11     xe_mmio_init(&root_tile->mmio, root_tile, xe->mmio.regs, SZ_4M);
12
13     return devm_add_action_or_reset(xe->drm.dev, mmio_fini, xe);
14 }
15
16 int xe_sriov_init(struct xe_device *xe) /* called from probe() */
17 {
18     [...]
19
20     xe->sriov.wq = alloc_workqueue("xe-sriov-wq", WQ_PERCPU, 0);
21     if (!xe->sriov.wq)
22         return -ENOMEM;
23
24     return drmm_add_action_or_reset(&xe->drm, fini_sriov, xe);
25 }
```

Xe — Lifecycle and Ownership

```
9     }
10
11     xe_mmio_init(&root_tile->mmio, root_tile, xe->mmio.regs, SZ_4M);
12
13     return devm_add_action_or_reset(xe->drm.dev, mmio_fini, xe);
14 }
15
16 int xe_sriov_init(struct xe_device *xe) /* called from probe() */
17 {
18     [...]
19
20     xe->sriov.wq = alloc_workqueue("xe-sriov-wq", WQ_PERCPU, 0);
21     if (!xe->sriov.wq)
22         return -ENOMEM;
23
24     return drmm_add_action_or_reset(&xe->drm, fini_sriov, xe);
25 }
26
27 int xe_gt_sriov_pf_control_init(struct xe_gt *gt) /* called from probe */
28 {
29     struct xe_device *xe = gt_to_xe(gt);
30
31     [...]
32 }
```

Xe — Lifecycle and Ownership

```
19
20     xe->sriov.wq = alloc_workqueue("xe-sriov-wq", WQ_PERCPU, 0);
21     if (!xe->sriov.wq)
22         return -ENOMEM;
23
24     return drmm_add_action_or_reset(&xe->drm, fini_sriov, xe);
25 }
26
27 int xe_gt_sriov_pf_control_init(struct xe_gt *gt) /* called from probe */
28 {
29     struct xe_device *xe = gt_to_xe(gt);
30
31     [...]
32
33     INIT_WORK(&gt->sriov.pf.control.worker, control_worker_func);
34     return drmm_add_action_or_reset(&xe->drm, control_fini_action, gt);
35 }
36
37 // Callchain 1: GuC CT path
38 //
39 // control_worker_func()
40 //   pf_worker_find_work()
41 //     pf_process_vf_state_machine()
42 //       pf_exit_vf_flr_send_start()
43 //         pf_send_vf_flr_start()
```

Xe — Lifecycle and Ownership

```
31     [...]  
32  
33     INIT_WORK(&gt->sriov.pf.control.worker, control_worker_func);  
34     return drmm_add_action_or_reset(&xe->drm, control_fini_action, gt);  
35 }  
36  
37 // Callchain 1: GuC CT path  
38 //  
39 // control_worker_func()  
40 //     pf_worker_find_work()  
41 //         pf_process_vf_state_machine()  
42 //             pf_exit_vf_flr_send_start()  
43 //                 pf_send_vf_flr_start()  
44 //                     pf_send_vf_control_cmd()  
45 //                         guc_action_vf_control_cmd()  
46 //                             xe_guc_ct_send_block()  
47 //                                 xe_guc_ct_send_recv()  
48 //                                     guc_ct_send()  
49 //                                         guc_ct_send_locked()  
50 //                                             __guc_ct_send_locked() ← returns -ECANCELED if CT stopped  
51 //                                                 h2g_write()  
52 //                                                     xe_map_memcpy_to() ← writes to CT ring (VRAM on dGPU)  
53 //                                                         xe_guc_notify()  
54 //                                                             xe_mmio_write32() ← doorbell register write
```

Conclusion

- Decouple lifetimes (and ownership) of userspace and driver state at the subsystem level.
- No more (subsystem) callbacks into the driver after driver unbind.
- Driver private (hardware specific) structures should be fully cleaned up on driver unbind.



Thank you!

Mailing list	nova-gpu@lists.linux.dev
Tree	gitlab.freedesktop.org/drm/rust/kernel.git drm-rust-next
Issues	gitlab.freedesktop.org/drm/nova/-/issues
Web	rust-for-linux.com/nova-gpu-driver
Docs	Documentation/gpu/nova/
Chat	rust-for-linux.zulipchat.com