# Rust in V4L2

## Where are we at?

Daniel Almeida

Consultant Software Engineer

daniel.almeida@collabora.com

* why ? *

# Why?

- Rust has better, more expressive ergonomics and types (Vec, Box, etc)
- Rust has a thriving community
- Native Rust is safe and fast

Most importantly: **some kernel communities are already experimenting successfully.**

but

There's a catch-22 here :/

Would you write a driver in Rust if you had to do all the infrastructure yourself?

## What do I hope to achieve here?

Consensus on **how** to move forward with the **low-risk, low-hanging fruit**

Let’s see what we have so far

# Some POD types

# A *very* thin videobuf2 abstraction

# Abstractions for *some* VIDIOC_* ioctls

The necessary code to get the driver to probe

A module that prints to the terminal
when processing some VIDIOC_* ioctls

```
rust/bindings/bindings_helper.h         |   8 +
rust/kernel/lib.rs                      |   2 +
rust/kernel/media/mod.rs                |   6 +
rust/kernel/media/v4l2/capabilities.rs  |  80 ++++
rust/kernel/media/v4l2/dev.rs           | 369 +++++++++++++++
rust/kernel/media/v4l2/device.rs        | 115 +++++
rust/kernel/media/v4l2/enums.rs         | 135 ++++++
rust/kernel/media/v4l2/format.rs        | 178 ++++++++
rust/kernel/media/v4l2/framesize.rs     | 176 +++++++
rust/kernel/media/v4l2/inputs.rs        | 104 +++++
rust/kernel/media/v4l2/ioctls.rs        | 608 +++++++++++++++++++++++++
rust/kernel/media/v4l2/mmap.rs          |  81 ++++
rust/kernel/media/v4l2/mod.rs           |  13 +
rust/kernel/media/videobuf2/core.rs     | 552 ++++++++++++++++++++++
rust/kernel/media/videobuf2/mod.rs      |   5 +
rust/kernel/sync.rs                     |   1 +
rust/kernel/sync/ffi_mutex.rs           |  70 +++
samples/rust/Kconfig                    |  11 +
samples/rust/Makefile                   |   1 +
samples/rust/rust_v4l2.rs               | 403 ++++++++++++++++
20 files changed, 2918 insertions(+)
```

# How does it work?

# General idea

- bindings_helper.h
- Raw bindings (bindings_generated.rs)
- Safe abstraction (rust/subsystem/foo.rs?)
- Actual driver (subsystem/usual_location/driver.rs)

# bindings_helper.h

```c
/* SPDX-License-Identifier: GPL-2.0 */
/*
 * Header that contains the code (mostly headers) for which Rust bindings
 * will be automatically generated by `bindgen`.
 *
 * Sorted alphabetically.
 */

#include <kunit/test.h>
#include <linux/amba/bus.h>
#include <linux/cdev.h>
#include <linux/clk.h>
#include <linux/errname.h>
#include <linux/file.h>
#include <linux/fs.h>
#include <linux/fs_parser.h>
#include <linux/gpio/driver.h>
#include <linux/hw_random.h>
/* more headers... */
```

Bindgen will process this to generate

**bindings_generated.rs**

```rust
pub struct vb2_buffer {
    pub vb2_queue: *mut vb2_queue, // Raw pointer (is this valid?)
    pub index: core::ffi::c_uint, // Regular integer vs Enum
    pub type_: core::ffi::c_uint,
    pub memory: core::ffi::c_uint,
    pub num_planes: core::ffi::c_uint,
    pub timestamp: u64_,
    pub request: *mut media_request, // Raw pointer
    pub req_obj: media_request_object, // This contains raw pointers
    pub state: vb2_buffer_state, // Type alias for c_uint
    pub _bitfield_1: __BindgenBitfieldUnit<[u8; 1usize], u8>, // Bitfield (unsafe)
    pub planes: [vb2_plane; 8usize], // Are all entries valid?
    pub queued_entry: list_head, // Shouldn't be exposed directly
    pub done_entry: list_head,
}
```

You can see that this struct has obvious issues

```rust
pub struct vb2_buffer {
    pub vb2_queue: *mut vb2_queue, // Raw pointer (is this valid?)
    pub index: core::ffi::c_uint, // Regular integer vs Enum
    pub type_: core::ffi::c_uint,
    pub memory: core::ffi::c_uint,
    pub num_planes: core::ffi::c_uint,
    pub timestamp: u64_,
    pub request: *mut media_request, // Raw pointer
    pub req_obj: media_request_object, // This contains raw pointers
    pub state: vb2_buffer_state, // Type alias for c_uint
    pub _bitfield_1: __BindgenBitfieldUnit<[u8; 1usize], u8>, // Bitfield (unsafe)
    pub planes: [vb2_plane; 8usize], // Are all entries valid?
    pub queued_entry: list_head, // Shouldn't be exposed directly
    pub done_entry: list_head,
}
```

Let’s have a look at what “vb2_ops”

translates to

```rust
pub struct vb2_ops {
    pub queue_setup: ::core::option::Option<
        unsafe extern "C" fn( // C linkage, will be called by the kernel directly
            q: *mut vb2_queue, // Receives the pointer from C, C manages the lifetime
            num_buffers: *mut core::ffi::c_uint,
            num_planes: *mut core::ffi::c_uint,
            sizes: *mut core::ffi::c_uint,
            alloc_devs: *mut *mut device,
        ) -> core::ffi::c_int, // Returns an int to the C side
    >,
    pub wait_prepare: ::core::option::Option<unsafe extern "C" fn(q: *mut vb2_queue)>,
    pub wait_finish: ::core::option::Option<unsafe extern "C" fn(q: *mut vb2_queue)>,
    pub buf_out_validate:
        ::core::option::Option<unsafe extern "C" fn(vb: *mut vb2_buffer) -> core::ffi::c_int>,
```

With C linkage, the kernel will be calling Rust directly. That’s where we get control.

Again, this is not suitable for general use. We must **bridge** this to a saf(er) API

How do we go about creating this saf(er) API?

Let’s look at the **trivial case**: plain “data” types

```rust
/// A wrapper over a pointer to `struct v4l2_capability`.
#[derive(Copy, Clone, Debug, PartialEq, PartialOrd)]
pub struct CapabilitiesRef(*mut bindings::v4l2_capability);

impl CapabilitiesRef {
    /// # Safety
    /// The caller must ensure that `ptr` is valid and remains valid for the lifetime of the
    /// returned [`CapabilitiesRef`] instance.
    pub unsafe fn from_ptr(ptr: *mut bindings::v4l2_capability) -> Self {
        Self(ptr)
    }

    // For internal convenience only.
    fn as_mut(&mut self) -> &mut bindings::v4l2_capability {
        // SAFETY: ptr is safe during the lifetime of [`CapabilitiesRef`] as per
        // the safety requirement in `from_ptr()`
        unsafe { self.0.as_mut().unwrap() }
    }

    /// Sets the `driver` field.
    pub fn set_driver(&mut self, driver: &[u8]) {
        let this = self.as_mut();
        let len = core::cmp::min(driver.len(), this.driver.len());
        this.driver[0..len].copy_from_slice(&driver[0..len]);
    }

    /// Sets the `card` field.
    pub fn set_card(&mut self, card: &[u8]) {
        let this = self.as_mut();
        let len = core::cmp::min(card.len(), this.card.len());
        this.card[0..len].copy_from_slice(&card[0..len]);
    }

    /// Sets the `bus_info` field.
    pub fn set_bus_info(&mut self, bus_info: &[u8]) {
        let this = self.as_mut();
        let len = core::cmp::min(bus_info.len(), this.bus_info.len());
        this.bus_info[0..len].copy_from_slice(&bus_info[0..len]);
    }
}
```

# How is this safe?

- Remember that pointer dereference is unsafe
- Assume that the pointer passed in by the kernel is valid
- Assume it remains valid for the lifetime of &self
- Under the above conditions, it’s OK to dereference it and modify it

Other types are harder: they also expose behavior

For these types, we must expose safe Rust functions/traits and call C behind the scenes

So much for that.

Let’s discuss something important

before we continue.

# Lifetimes!

Let’s have a look at that trivial abstraction again

```rust
/// A wrapper over a pointer to `struct v4l2_capability`.
#[derive(Copy, Clone, Debug, PartialEq, PartialOrd)]
pub struct CapabilitiesRef(*mut bindings::v4l2_capability);

impl CapabilitiesRef {
    /// # Safety
    /// The caller must ensure that `ptr` is valid and remains valid for the lifetime of the
    /// returned [`CapabilitiesRef`] instance.
    pub unsafe fn from_ptr(ptr: *mut bindings::v4l2_capability) -> Self {
        Self(ptr)
    }

    // For internal convenience only.
    fn as_mut(&mut self) -> &mut bindings::v4l2_capability {
        // SAFETY: ptr is safe during the lifetime of [`CapabilitiesRef`] as per
        // the safety requirement in `from_ptr()`
        unsafe { self.0.as_mut().unwrap() }
    }

    /// Sets the `driver` field.
    pub fn set_driver(&mut self, driver: &[u8]) {
        let this = self.as_mut();
        let len = core::cmp::min(driver.len(), this.driver.len());
        this.driver[0..len].copy_from_slice(&driver[0..len]);
    }

    /// Sets the `card` field.
    pub fn set_card(&mut self, card: &[u8]) {
        let this = self.as_mut();
        let len = core::cmp::min(card.len(), this.card.len());
        this.card[0..len].copy_from_slice(&card[0..len]);
    }

    /// Sets the `bus_info` field.
    pub fn set_bus_info(&mut self, bus_info: &[u8]) {
        let this = self.as_mut();
        let len = core::cmp::min(bus_info.len(), this.bus_info.len());
        this.bus_info[0..len].copy_from_slice(&bus_info[0..len]);
    }
}
```

Note that we get a *pointer*, IOW: **the kernel manages the lifetime**

# Lifetime for C objects

- Again, **the kernel controls the lifetime**
- There's no relationship between our wrapper dropping and the C object being cleaned
- **But we can use it when the kernel passes it to us**

Wait, if C is controlling the lifetime in a lot of cases, why are we doing this then?

Let's get some things out of the way here:

Yes, this is only as safe as the bindings are safe

But, even when the kernel controls the lifetime, we get the following benefits

- We get to benefit from Rust’s ergonomics
- We get to benefit from the types in core::*
- Other Rust features still apply (i.e. the reference rules and guarantees still apply)
- There’s a subset of the kernel that *really* benefits from the above

But most importantly, we must break the catch-22 here

# Why?

- **This is an investment**: it paves the way for new kernel frameworks to be written in Rust from the ground up
- With native Rust, that’s where we start to reap some *major* benefits

# Where can we start?

# Some low-hanging fruit

- Codec libraries and parsers (VP9, AV1, JPEG, H.264, etc)
- The codec-specific logic in codec drivers (e.g. writing codec metadata to MMIO registers)

This offers a low-risk path for us to experiment with Rust

# Some roadblocks

- Maintainership issues
- The huge amount of work involved in abstractions
- Issues in the C code itself
- Not everybody knows Rust
- Will this break existing C code?

# Maintainership issues

- Well, I volunteer
- I expect people benefitting from Rust to help out as we go
- IMHO, the most important thing is to notice whether a change should touch the Rust side
- I wonder if we can automate the above

# The “huge amount of work” issue

- We do not have to create bindings for every thing under the sun
- Only the “entrypoints” should have bindings, i.e. only things directly called by drivers
- And even still, we only need to write these as we see the need for them

# Issues in the C code itself

Whatever issues with C will be fixed by proxy whenever the C code is fixed

No, this will *not* break existing C code,

how could it?

# Miguel Ojeda's suggestion

```
> Some subsystems may want to give that maintainer a different
> `MAINTAINERS` entry, e.g. as a child subsystem that sends PRs to the
> main one and may be marked as "experimental". This is also a way to
> see how the new abstractions work or not, giving maintainers more time
> to decide whether to commit to a Rust side or not
```

# Summary

- Yes, this will be hard
- There are ways we can make this less risky and move along if it fails
- IMHO we should try this, it might work out :)

# By the way

- We can use proc_macros for the “plain data” types to write the boilerplate
- I am working on just enough bindings so I can write a barebones visl clone (stateless m2m decoder)

Thoughts?