# From C Ops-Tables to Safe Rust Traits

## Interop Patterns for `no_std` Embedded

**Akshai M** · Analog Devices

1st Ariel OS Community Day · Grenoble · 1 Sep 2026

## About me

- **Akshai M**, System Architect, Software Defined Vehicles Enablement (Analog Devices)
- Path: a brief stint with RIOT OS, then STABL Energy, now Analog Devices
- Focus: Rust in automotive and industrial embedded, with a strong emphasis on safety and reliability

## The starting point

- Years of **proven C drivers** already exist and ship in products
- We want **Rust**: memory safety, better APIs, no data races
- A full **rewrite is not realistic**: cost, risk, regressions
- Target is deeply embedded: `no_std` **, no heap**

> Can we get Rust's safety **on top of existing C drivers**, at **zero runtime cost**?

ANALOG DEVICES

## Constraints we set

- `no_std`, **allocation-free** core
- **Reuse the ecosystem** (`embedded-hal`, `embedded-io`) instead of reinventing buses
- **Zero-cost** by default
- Support **runtime-pluggable** drivers (chosen or loaded at runtime, maybe across an ABI)
- Keep `unsafe` **small and auditable**

# How C ships pluggable drivers

```
struct adc_ops {
    int (*power_on)(struct adc_handle *h);
    int (*set_vref)(struct adc_handle *h, enum vref_src src);
    int (*read)(struct chan_handle *h, void *data);
    /* ... more ops ... */
};
```

- A **struct of function pointers** = a vtable
- **Opaque handles** (`void*`-style) hide driver state

## The Rust side we build on

- **`embedded-hal`** for buses (SPI, I2C): uses `&[u8]`, so **reuse it**
- **`embedded-io`** for byte streams (UART): complementary, not a substitute
- **Define only what's missing**: device classes (e.g. an ADC)
- Error model mirrors `embedded-hal`: `ErrorType` / `Error` / `ErrorKind`

## The core question

How do we expose a C ops-table of raw function pointers and opaque handles as a safe, idiomatic Rust trait, without giving up zero cost or `no_std`?

## Start with the trait (the *what*)

```
pub trait AdcDevice: ErrorType {
    fn enable(&mut self) -> Result<(), Self::Error>;
    fn set_vref(&mut self, src: VrefSource) -> Result<(), Self::Error>;
    fn convert(&mut self, chan: u8) -> Result<u32, Self::Error>;
}
```

- A **hardware-agnostic contract**, the vocabulary that apps speak
- Same idea as the C ops-table, but **compiler-checked** and **zero-cost**

# The bridge (the *how*)

```rust
#[repr(C)]                                  // use C's memory layout, matching the C struct
struct AdcOps {                             // the ops table: a set of function pointers
    power_on: unsafe extern "C" fn(*mut c_void) -> i32,
    // one field per operation
}

struct ForeignAdc<'a> {                     // the safe wrapper around the C driver
    ops: &'static AdcOps,                   // the table of function pointers
    ctx: *mut c_void,                       // the driver's private state, an opaque handle
    _p: PhantomData<&'a mut ()>,            // ties the wrapper to a borrow of that state
}

impl AdcDevice for ForeignAdc<'_> {  // present the raw table as a safe Rust trait
    fn enable(&mut self) -> Result<(), Self::Error> {
        // SAFETY: ctx is a valid context for ops, guaranteed at construction
        // call the function pointer, then turn its status code into a Result
        check(unsafe { (self.ops.power_on)(self.ctx) })
    }
}
```

Every trait call becomes a call through a C function pointer.

## Confine the `unsafe`

- `extern "C"` fn pointers carry preconditions, so calling them is `unsafe`
- **Wrap once** and expose a safe trait, so callers write **zero** `unsafe`
- Prefer a **safe, lifetime-bounded constructor** when the storage is Rust-owned

```
// before:  let adc = unsafe { ForeignAdc::new(&OPS, raw_ptr) };
// after:   let adc = bind(&mut state);   // no unsafe in the app
```

# Dispatch: the menu

| Strategy | What it is |
|---|---|
| **Generics** (monomorphization) | static, zero-cost |
| `dyn Trait` | in-process runtime dispatch (Rust vtable, unstable ABI) |
| **Enum dispatch** | closed, compile-time-known set |
| `#[repr(C)]` **ops-table** | stable **C ABI** (C drivers, separate compilation; `dlopen` on hosted targets) |

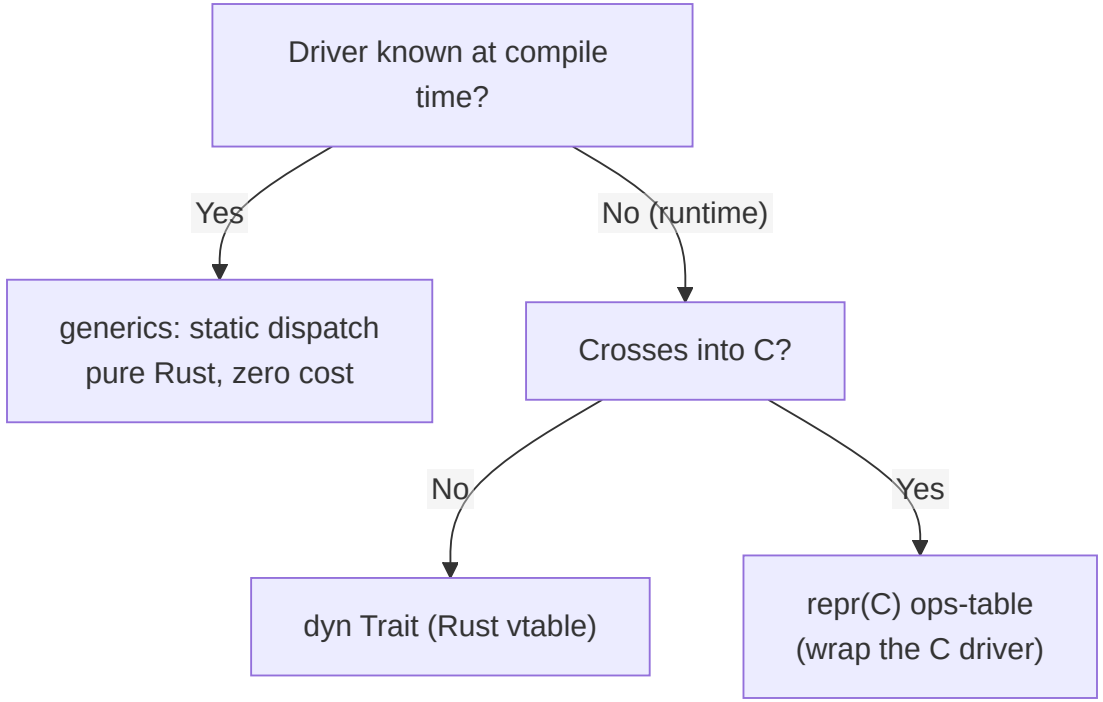
# What does each cost?

**Single dispatch, Cortex-M4 (`thumbv7em-none-eabihf`), `-O`**

| Strategy | Code size | Instructions |
|---|---|---|
| C ops-table (baseline) | 64 B | 23 |
| Rust generics (static) | **16 B** | **7** |
| Rust `dyn` | 54 B | 23 |
| Rust enum | 42 B | 18 |
| Rust `#[repr(C)]` ops-table | 56 B | 24 |

- **Static wins** when the driver is known at compile time
- **`#[repr(C)]` ops-table ≈ C's own cost**, and gives cross-ABI pluggability

# Choosing the mechanism per driver




> Static by default. Pluggable only where you actually need it.

## An incremental path to full Rust

Wrap today's C drivers first, then migrate one driver at a time. Each one moves the whole stack toward pure Rust:

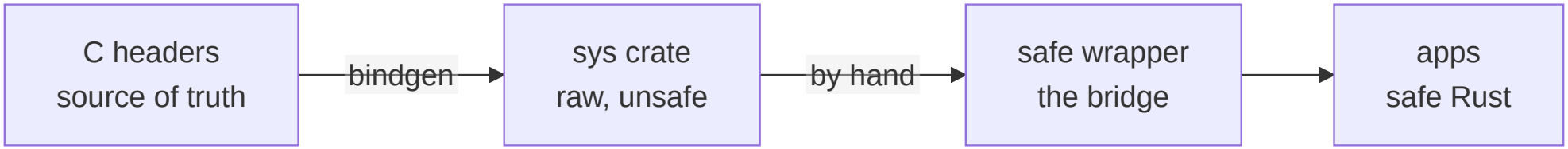
# Who owns the ABI?

Two **different** sources of truth:

| Layer | Owned by |
|---|---|
| The **C ABI** (ops-table, handles, enums) | the **C headers** |
| The **ergonomic Rust API** (trait, typestate) | **Rust** (a design choice) |

- Given a mature C code base, **don't re-declare the ABI by hand**, which invites drift
- Generate the raw layer from the headers

# The `-sys` + safe-wrapper architecture

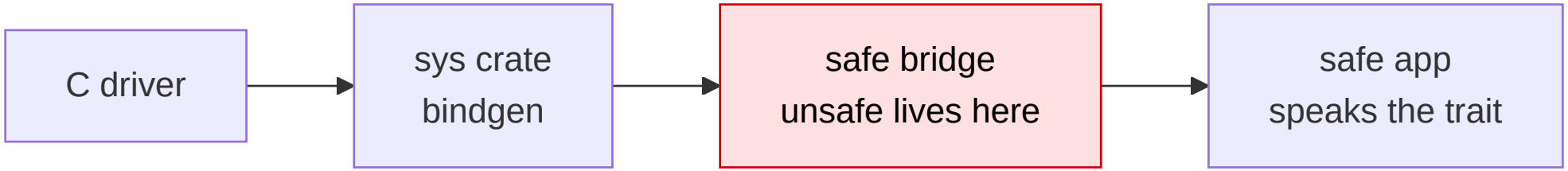



- **bindgen** generates the **type declarations** (shapes), *not* the bridge
- This is exactly the **Rust-for-Linux** pattern (`bindings` + `kernel` + `helpers.c`)

## Generating the glue

- The bridge boilerplate can be produced by a **vtable-style proc-macro**
- Rust-for-Linux's `#[vtable]`: `HAS_<method>` flags, **`NULL` for absent optional ops**
- The struct **type still comes from bindgen**; the macro fills it + trampolines

# The whole picture




- `unsafe` lives **only** in the bridge
- Swap the C driver and the app doesn't change a line

# Lessons learned

- **Reuse** `embedded-hal` / `embedded-io`; define only what's missing
- A `#[repr(C)]` **ops-table** is the zero-overhead, cross-ABI dispatch option
- **Confine** `unsafe` to a thin, audited boundary
- Let the **C headers stay the single source of truth**; generate the raw layer
- **Static dispatch when you can, pluggable when you must**

## Takeaways

- A trait is the safe, zero-cost, compiler-checked version of a C ops-table
- You can wrap large existing C drivers **without a rewrite** and **without a heap**
- Applicable to any `embedded-hal` **/ Embassy-based** environment

## Thank you. Questions?

Akshai M · akshai.m@analog.com