

Creating self references safely

Gary Guo

Linux Plumbers Conference 2026

07/10/2026, Prague

Problem statement

Device resource lifetimes

Devices have resources, and the resources may only be accessed when the device is **bound**.

```
struct MyDriver {
    dev: ARef<Device>,
    bar: Arc<Devres<Bar<BAR_SIZE>>>,
    irq: Devres<irq::Registration<MyIrqHandler>>,
}

struct MyIrqHandler {
    bar: Arc<Devres<Bar<BAR_SIZE>>>,
}

impl irq::Handler for MyIrqHandler {
    fn handle(&self) -> irq::IrqReturn {
        if let Some(bar) = self.bar.try_access() {
            let status = bar.read(regs::IRQ_STATUS);
        }
    }
}
```

Device resource lifetimes

Devres and failure path is gone, but Arc remains!

```
struct MyDriver<'bound> {  
    dev: &'bound Device,  
    bar: Arc<Bar<'bound, BAR_SIZE>>,  
    irq: irq::Registration<'bound, MyIrqHandler<'bound>>,  
}
```

```
struct MyIrqHandler<'bound> {  
    bar: Arc<Bar<'bound, BAR_SIZE>>,  
}
```

```
impl irq::Handler for MyIrqHandler<'_> {  
    fn handle(&self) -> irq::IrqReturn {  
        let status = self.bar.read(regs::IRQ_STATUS);  
    }  
}
```

The problem

We want static control flow over dynamic. If resource A depends on resource B, we want to ensure that resource A is destroyed first, and then B.

The 'bound lifetime solves the issue of dependency between device resources and device itself, but not between different resources.

Rust's lifetime allows you to represent relationship *between* objects. But it does not allow you to track relationship *within* an object!

What we want

```
struct MyDriver<'bound> {  
    dev: &'bound Device,  
    bar: Bar<'bound, BAR_SIZE>,  
    irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar>>,  
}
```

```
struct MyIrqHandler<'bound, 'bar> {  
    bar: &'bar Bar<'bound, BAR_SIZE>,  
}
```

Instead of reference counting, `MyIrqHandler` would hold a reference to `Bar` instead.

Good news

```
#[pin_data]
struct MyDriver<'bound> {
    dev: &'bound Device,
    bar: Bar<'bound, BAR_SIZE>,
    #[pin]
    irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar>>,
}

struct MyIrqHandler<'bound, 'bar> {
    bar: &'bar Bar<'bound, BAR_SIZE>,
}
```

Drop order check

pin-init detects wrong drop order at compile time!

```
error: field `irq` borrows `bar`, but drops later
```

```
→ samples/rust/rust_driver_demo.rs:55:55
```

```
55 |     irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar>>,  
    |                                     ^^^^
```

The fixed version

```
#[pin_data]
struct MyDriver<'bound> {
    dev: &'bound Device,
    #[pin]
    irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar>>,
    // Destroy this later!
    bar: Bar<'bound, BAR_SIZE>,
}

struct MyIrqHandler<'bound, 'bar> {
    bar: &'bar Bar<'bound, BAR_SIZE>,
}
```

Quick user guide

Struct declaration

```
#[pin_data]
struct MyDriver<'bound> {
    dev: &'bound Device,
    #[pin]
    irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar>>,
    bar: Bar<'bound, BAR_SIZE>,
}

struct MyIrqHandler<'bound, 'bar> {
    bar: &'bar Bar<'bound, BAR_SIZE>,
}
```

If you want to borrow from a field, just name the lifetime with the same name!

Initialization

```
fn probe<'bound>(dev: &'bound Device<Core<'_>>) ->
    impl PinInit<MyDriver<'bound>, Error>
{
    try_pin_init!(MyDriver {
        bar: dev.iomap_region_sized::<BAR_SIZE>(0, c"bar0")?,
        vectors: dev.alloc_irq_vectors(...)?,
        irq <- irq::Registration::new(
            vectors.index(0)?.into(),
            /* ... */,
            Ok(MyIrqHandler { bar })
        ),
    })
}
```

Everything happens automagically in initializer.

A let binding is created for borrowed fields and you can use it to initialize other fields.

Access

If you need to access the fields that reference other fields, some changes are needed.

```
fn access_irq(driver: Pin<&mut MyDriver<'bound>>) {  
    // Accessor method  
    driver.irq();  
  
    // Projection. Lifetime is shortened to that of `driver`  
    driver.project().irq;  
  
    // Newly added projection.  
    driver.with_project(|project| {  
        // Inside the closure things have proper lifetime.  
        // Code inside can correlate `project.irq` with `project.bar`.  
        project.irq;  
    });  
}
```

For demonstration only. In practice you do not access registration nor have a mutable access to driver data.

What does it not do

You cannot get a reference of the full struct still.

Pin-init self reference vs managed APIs

On C side, we have created many *managed* API (devm, drmm) to ensure that things are created and destroyed in proper order.

Why not always use managed APIs instead? For managed APIs, things are teared down in the reverse order of initialization.

Mismatch between initialization and destruction order

If everything is *immutable*, then the desired destruction order is the reverse of initialization. This is not always true when things are *mutable*.

Mismatch between initialization and destruction order

If everything is *immutable*, then the desired destruction order is the reverse of initialization. This is not always true when things are *mutable*.

```
struct MyDriver<'bound> {  
    dev: &'bound Device,  
    #[borrows(invariant 'bar)]  
    irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar>>,  
    bar: Bar<'bound, BAR_SIZE>,  
}
```

```
struct MyIrqHandler<'bound, 'bar> {  
    bar: SpinLock<Option<&'bar Bar<'bound, BAR_SIZE>>>,  
}
```

Mismatch between initialization and destruction order

If everything is *immutable*, then the desired destruction order is the reverse of initialization. This is not always true when things are *mutable*.

```
try_pin_init!(MyDriver {  
    irq <- irq::Registration::new(  
        vectors.index(0)?.into(), /* ... */  
        try_pin_init!(MyIrqHandler {  
            bar <- new_spinlock!(None),  
        }),  
    ),  
    bar: dev.iomap_region_sized::<BAR_SIZE>(0, c"bar0"?),  
})
```

// Later

```
*irq.handler().bar.lock() = Some(bar);
```

Mismatch between initialization and destruction order

When initialization fails, pin-init still destroys things in reverse initialization order, consistent with Rust scope rules.

```
try_pin_init!(MyDriver {  
    irq <- irq::Registration::new(  
        vectors.index(0)?.into(), /* ... */  
        try_pin_init!(MyIrqHandler {  
            bar <- new_spinlock!(None),  
        }),  
    ),  
    bar: dev.iomap_region_sized::<BAR_SIZE>(0, c"bar0"?),  
    -: {  
        *irq.handler().bar.lock() = Some(bar);  
        Err(EINVAL)?  
    }  
})
```

Mismatch between initialization and destruction order

pin-init checks the data dependency inconsistency cannot appear.

```
error[E0505]: cannot move out of value because it is borrowed
  → /home/gary/Projects/linux/samples/rust/rust_driver_demo.rs:96:9
88 |         irq ← unsafe { irq::Registration::new(
   |         --- borrow later used here
...
96 |         bar: dev.iomap_region_sized::<BAR_SIZE>(0, c"bar0")?,
   |         ^^^
   |         |
   |         move out of value occurs here
   |         borrow of value occurs here
   = note
: this error originates in the macro `::pin_init::pin_init` which comes from the expansion of the macro
`try_pin_init` (in Nightly builds, run with -Z macro-backtrace for more info)
```

What pin-init self-reference enables

You have full control of the order of initialization and destruction.

Pin-init will make sure that it is correct.

Variance annotation

In most cases, interior mutability is not needed, so the type should be covariant over the generated field lifetime.

This will be inferred by pin-init as what you want. If that is the case, things will just work™.

Explicit annotations can be used to override variance, or aid pin-init when it cannot detect certain usage.

```
#[pin_data]
struct MyStruct {
    // By default, covariance is implied (and checked)
    covariant: &'owner String,
    // You can put explicit annotation for cases where pin-init cannot detect.
    #[borrows(covariant 'owner)]
    explicit_covariant: hidden_behind_macro!(&'owner String),
    #[borrows(invariant 'owner)]
    not_covariant: Mutex<&'owner String>,
    owner: String,
}
```

Variance annotation

Variance is checked, so there is no need to worry about getting it wrong.

```
error: lifetime may not live long enough
→ samples/rust/rust_driver_demo.rs:58:8
53 | #[pin_data]
   | _____ in this procedural macro expansion
...
58 |     irq: irq::Registration<'vectors, MyIrqHandler<'bound, 'bar>>,
   |           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |           |
   |           lifetime `'_short` defined here
   |           lifetime `'_long` defined here

function was supposed to return data with lifetime `'_long` but it is returning data with lifetime
`'_short`
= help: consider adding the following bound: `'_short: 'long`
= note
: requirement occurs because of the type `MyIrqHandler<'_, ' _>`, which makes the generic argument `'_`
invariant
= note: the struct `MyIrqHandler<'bound, 'bar>` is invariant over the parameter `'_bound`
= help: see <https://doc.rust-lang.org/nomicon/subtyping.html> for more information about variance
```

Variance annotation

Bikeshedding: `#[borrows]` attribute – maybe `#[captures]` instead?

Mutable borrows

If you have a need to borrow a field mutably, add `#[borrowed(mut)]` annotation.

```
struct MyDriver<'bound> {  
    dev: &'bound Device,  
    irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar>>,  
    #[borrowed(mut)]  
    bar: Bar<'bound, BAR_SIZE>,  
}
```

Then you will be given a mutable reference in the initializer to store in a sibling field.

Borrowing from multiple fields

If you have a need to borrow from multiple fields, just let pin-init know.

```
struct MyDriver<'bound> {  
    dev: &'bound Device,  
    #[borrows('bar0, 'bar1)] // Tell pin-init you need both  
    irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar0>>,  
    bar0: Bar<'bound, BAR_SIZE>,  
    bar1: Bar<'bound, BAR_SIZE>,  
}
```

You can use the shortest living lifetime in the type, and pin-init will insert outlive bounds between lifetimes so your code builds.

Borrowing from multiple fields

Note that pin-init will only insert outlive bounds where it is sound to do so. So it is possible that bounds do not exist between invariant and covariant borrows. When that happens, apply variance annotations uniformly.

```
struct MyDriver<'bound> {  
    dev: &'bound Device,  
    #[borrows(invariant 'bar0, invariant 'bar1)]  
    irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar0>>,  
    bar0: Bar<'bound, BAR_SIZE>,  
    bar1: Bar<'bound, BAR_SIZE>,  
}
```

Existential lifetime

Invariance propagation

For complex type theory reasons, capturing a field lifetime invariantly can also cause lifetime referenced by that field to be invariant too.

```
struct MyDriver<'bound> {  
    dev: &'bound Device,  
    #[borrows(invariant 'bar)]  
    irq: irq::Registration<'bound, MyIrqHandler<'bound, 'bar>>,  
    bar: Bar<'bound, BAR_SIZE>,  
}
```

The type above will cause the type to be invariant over 'bound.

This is fine for driver data, but it might be undesirable in many other cases.

Invariance propagation

```
#[pin_data]
struct Cmdq<'a>
{
    #[pin]
    inner: Mutex<CmdqInner<'a>>,
    ...
}
```

Mutex is another source of invariance. The `Cmdq` needs access to device resources, so it needs to capture some lifetime.

Invariance is required so that you cannot put device resources of shorter-living lifetime into the `Mutex`, which may be destroyed later. However, mostly likely you do not need to put in a new device resource. Effectively, the device resources are not actually mutable inside `Mutex`, it just protects concurrent access (with some other data that is actually mutable).

Existential lifetime

Existential lifetime provides you a way to detach variance.

```
#[pin_data]
pub(crate) struct Cmdq<'a>
where
    exists<'dev>: 'a,
{
    #[pin]
    #[borrows(invariant 'dev)]
    inner: Mutex<CmdqInner<'dev>>,
    ...
}
```

Here, `'dev` is only required to outlive `'a`. The actual lifetime is erased.

Questions and Discussions