Improving resource ownership and life-time in linux device drivers

Bartosz Golaszewski

Linux Plumbers Conference 2023, Richmond VA

About

- Linux kernel developer for the Qualcomm Landing Team at Linaro
- 15 years of experience
- Maintainer of the GPIO subsystem
- Author and maintainer of libgpiod
- Interested in complex software architecture

Background

- Laurent Pinchart at Linux Plumbers 2022

  “Why is devm_kzalloc() harmful and what can we do about it”

  https://www.youtube.com/watch?v=kW8LHWIJPTU
- Bartosz Golaszewski at FOSDEM 2023

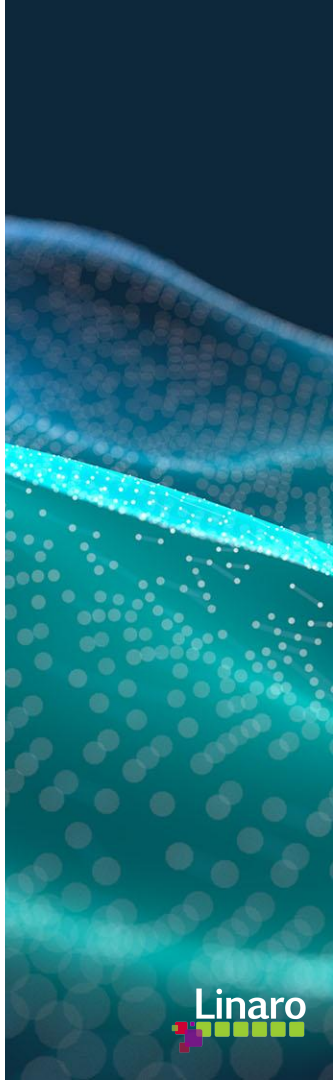
  “Don't blame devres - devm_kzalloc() is not harmful

  Use-after-free bugs in drivers and what to do about them.”

  https://archive.fosdem.org/2023/schedule/event/devm_kzalloc/
- Wolfram Sang at EOSS 2023

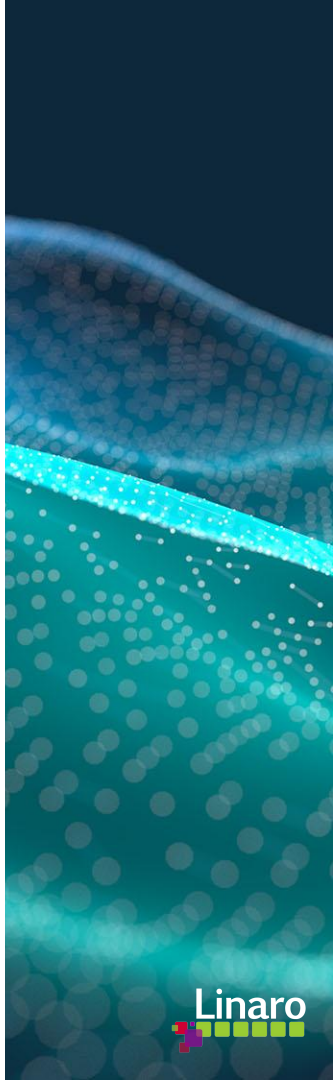
  “Subsystems with Object Lifetime Issues (in the Embedded Case)”

  https://www.youtube.com/watch?v=HCiJL7djGw8

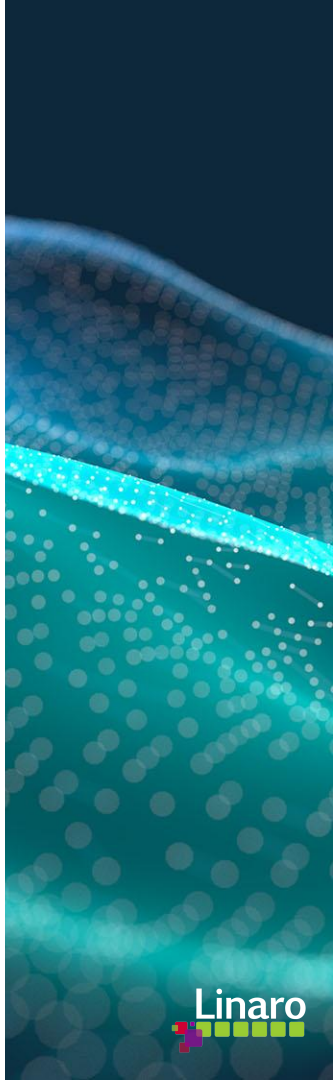


Background

- Using devres seemed to cause memory problems
- It's not really devres
- Drivers embed reference counted `struct device` objects within private data structures that are freed at driver detach
- Subsystems proceed to implement the wildest workarounds to not crash or just give up and outright explode if a driver is detached with references to `struct device` still held
- This problem is at least 18 years old and fixing it is hard

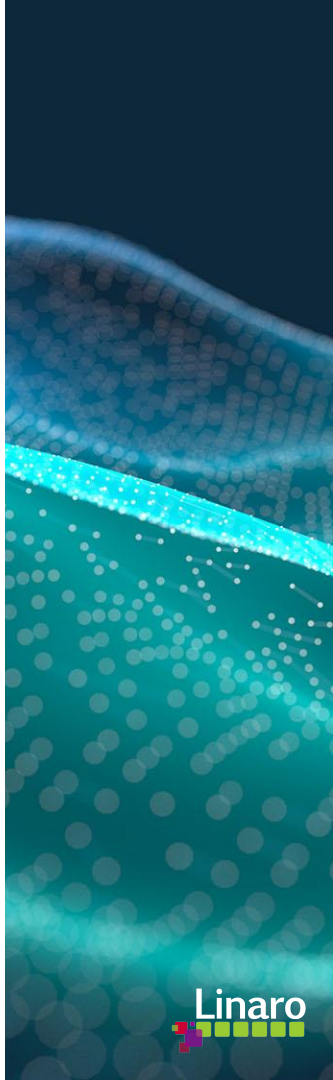


Agenda



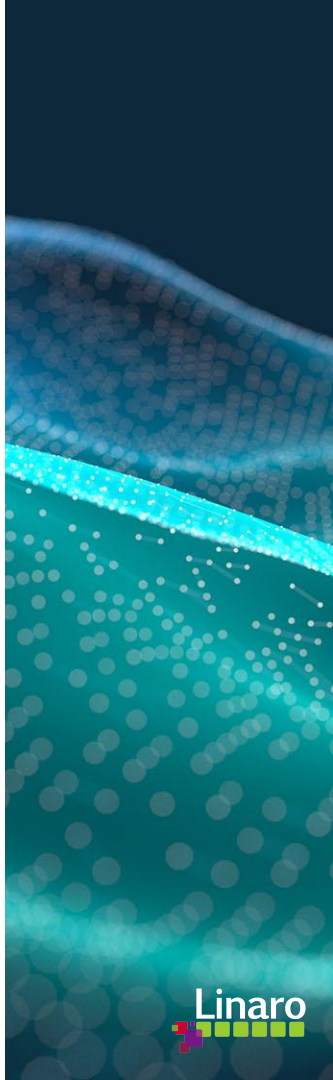
Agenda

- More stuff is broken



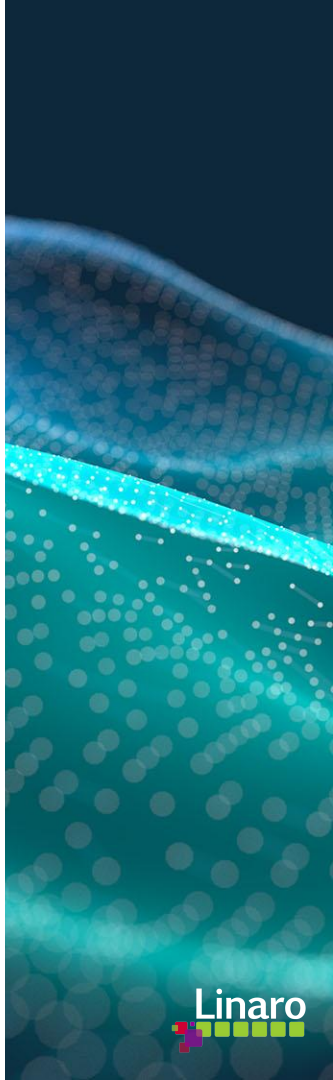
Glossary

- **Resource** - software representation of some hardware asset, e.g. regulator, nvmem cell, GPIO, clock, interrupt, reset
- **Resource handle** - concrete software data structure associated with an instance of a resource, e.g.: struct regulator, struct gpio_desc, int irq
- **Resource provider** - device (bound to its driver) in control of the hardware asset exposing resources to consumers and dealing out resource handles, e.g. regulator driver
- **Resource consumer** - device (bound to its driver) retrieving a resource handle from the provider e.g.: SPI controller driver requesting the chip-select GPIO

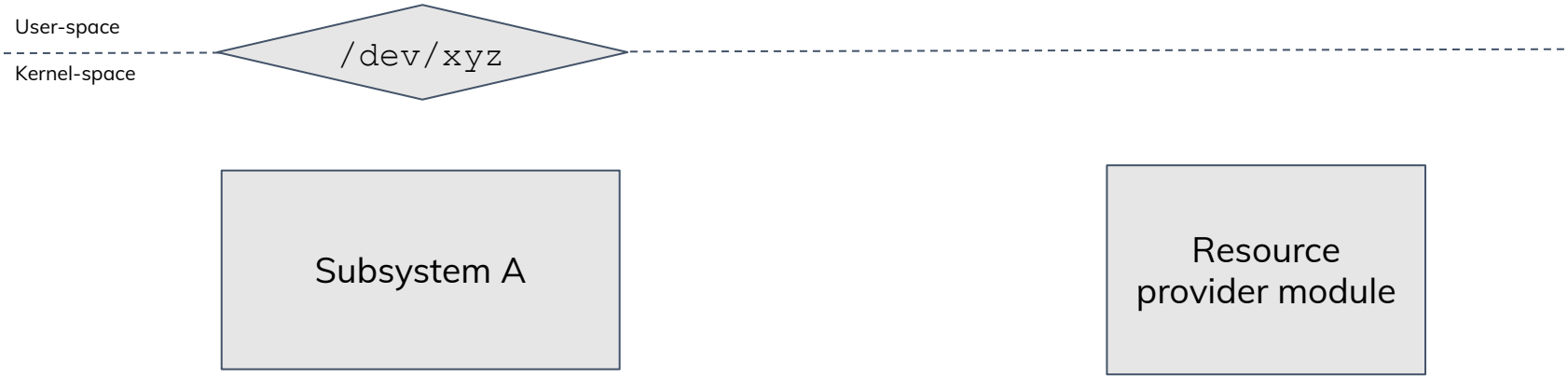


Kernel driver “subsystems”

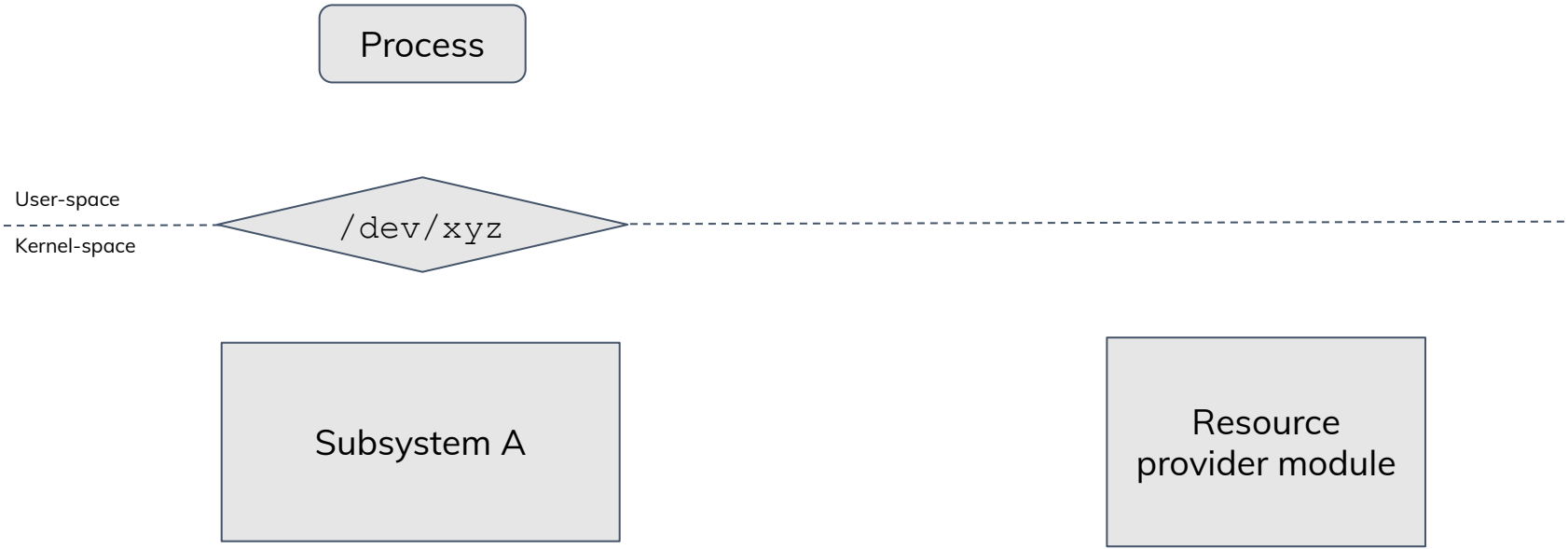
- Provide abstraction layers and enable code reuse for drivers
- Drivers often go through several abstraction layers
- It seems many subsystems have been developed by copying existing solutions which are not always correct
- In general not scrutinized as much as the core kernel code



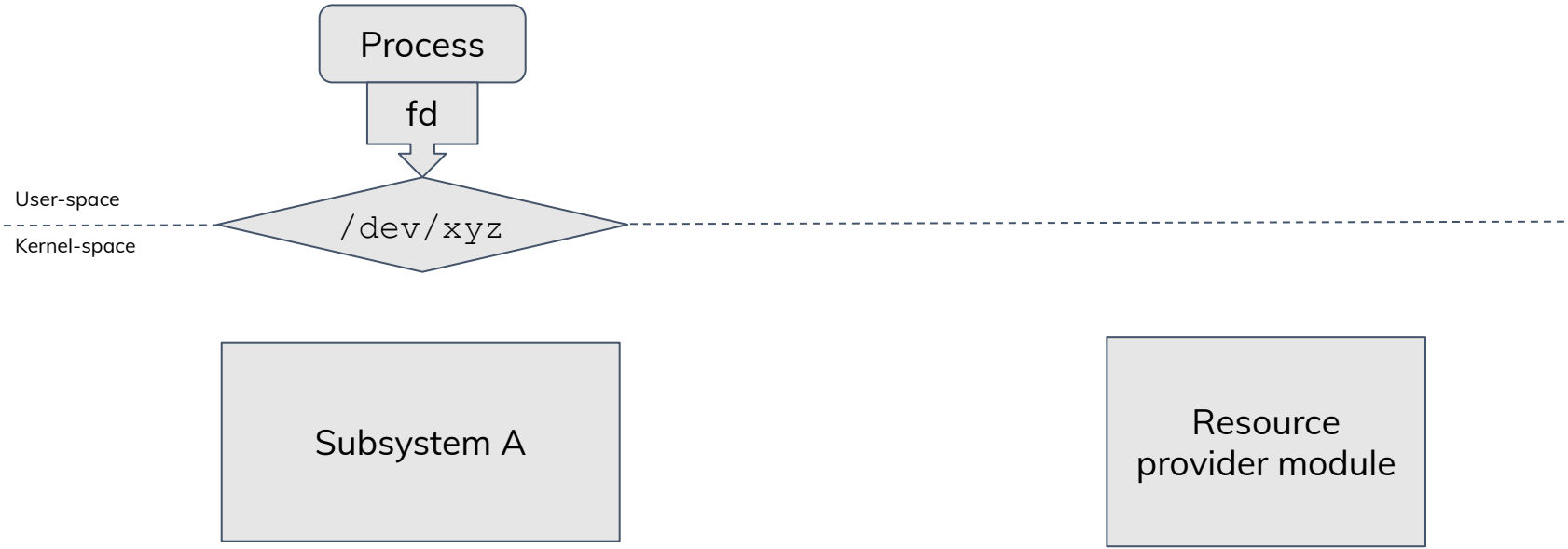
Thought experiment



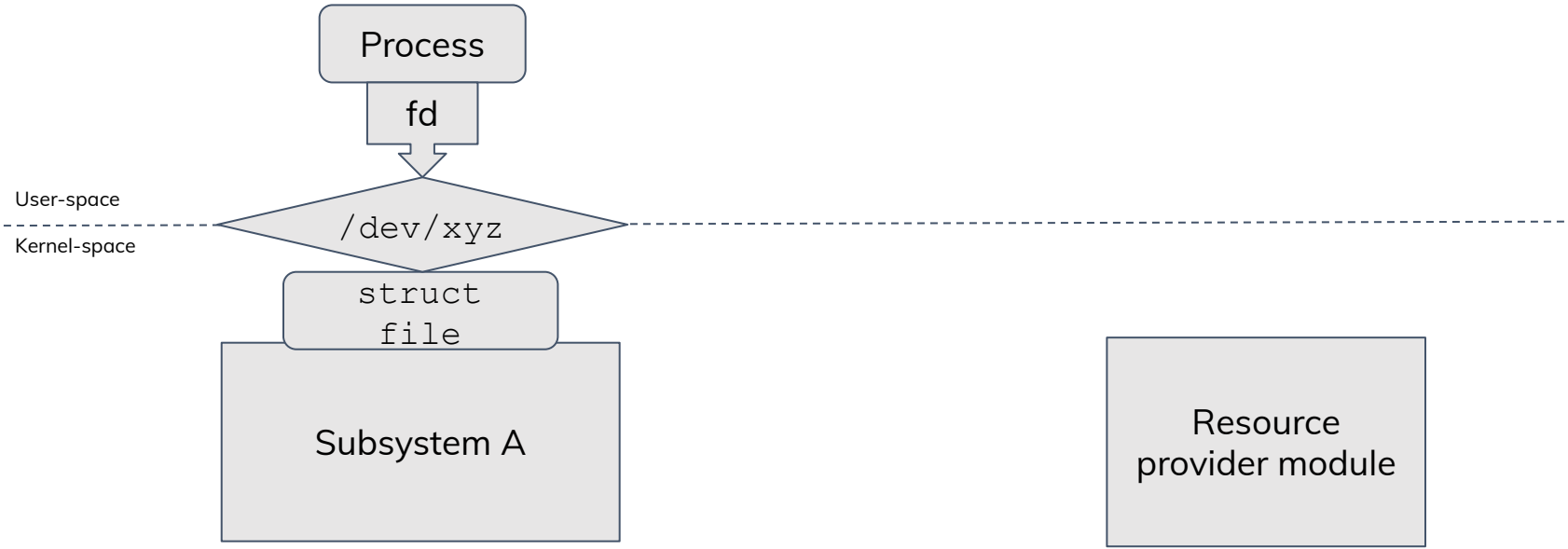
Thought experiment



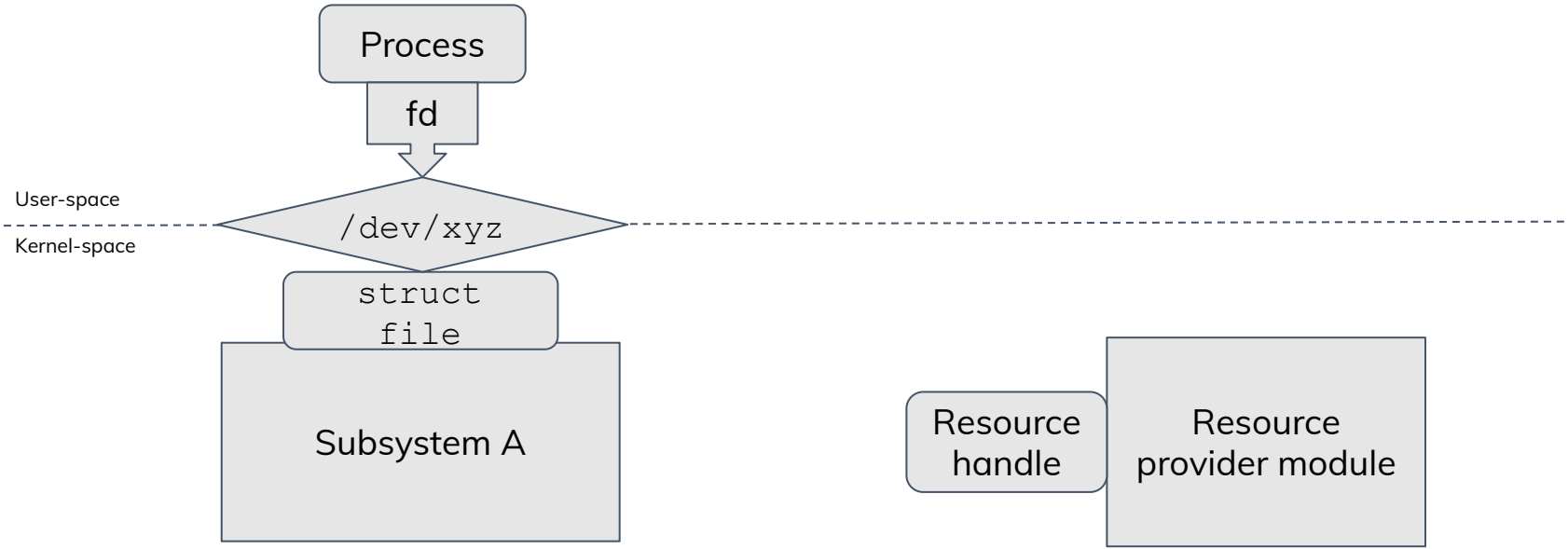
Thought experiment



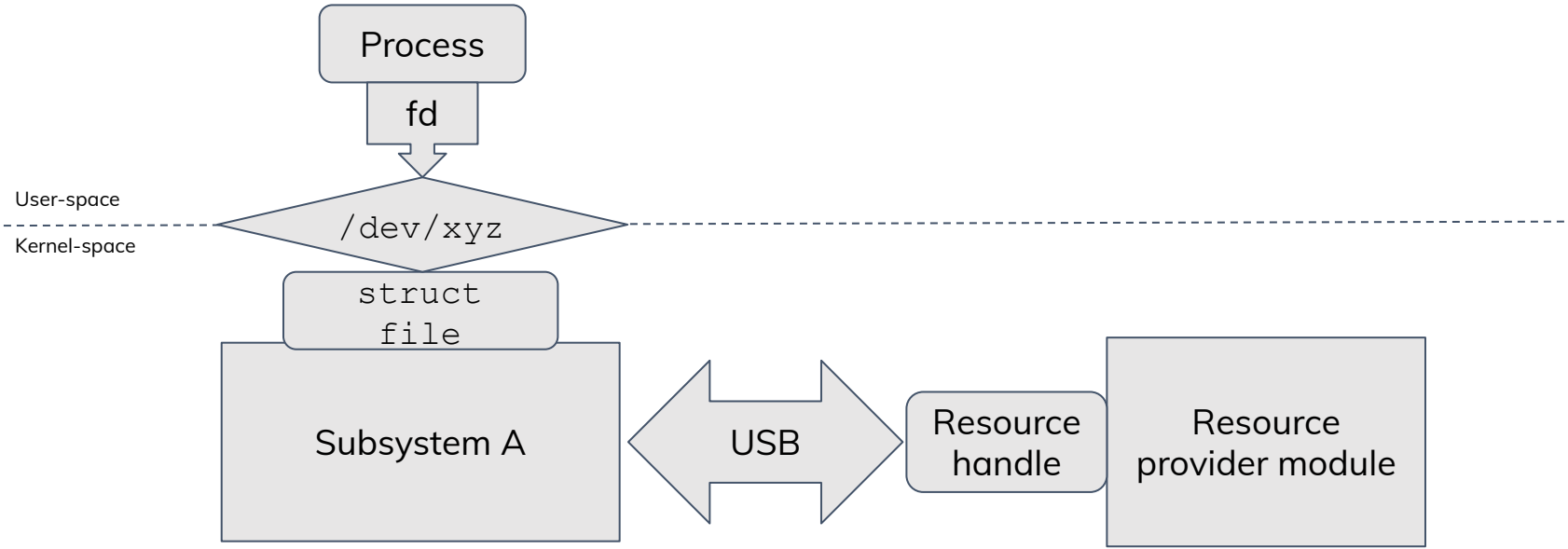
Thought experiment



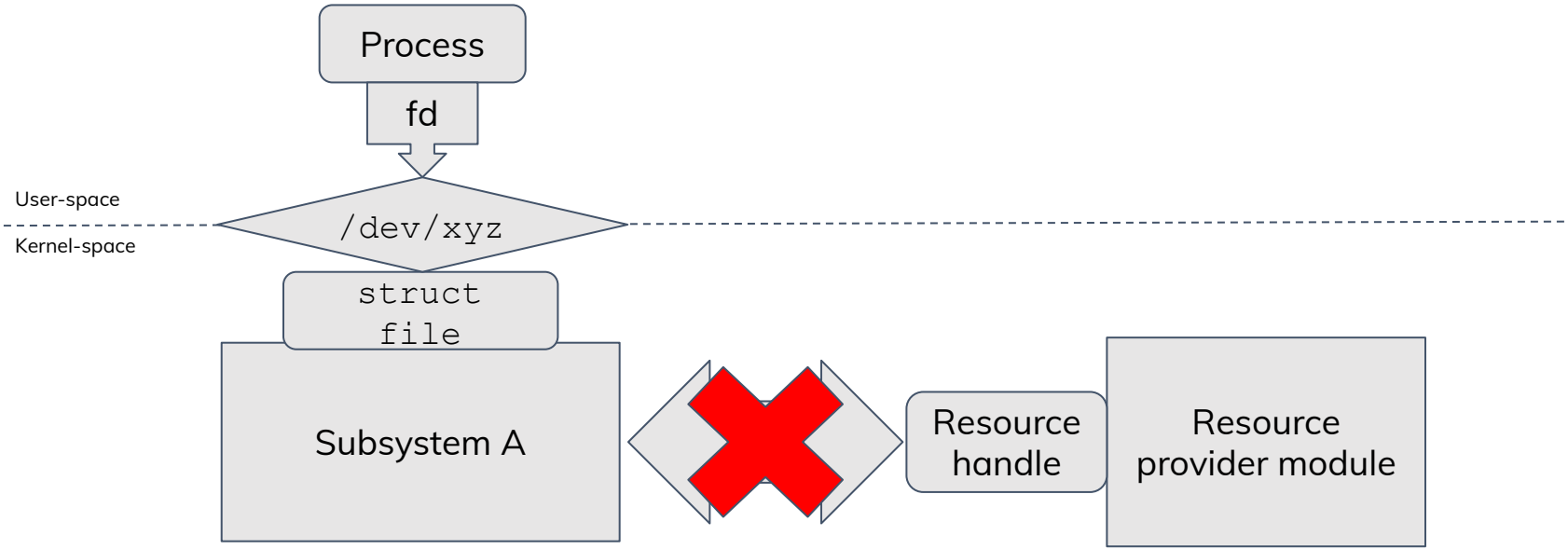
Thought experiment



Thought experiment



Thought experiment



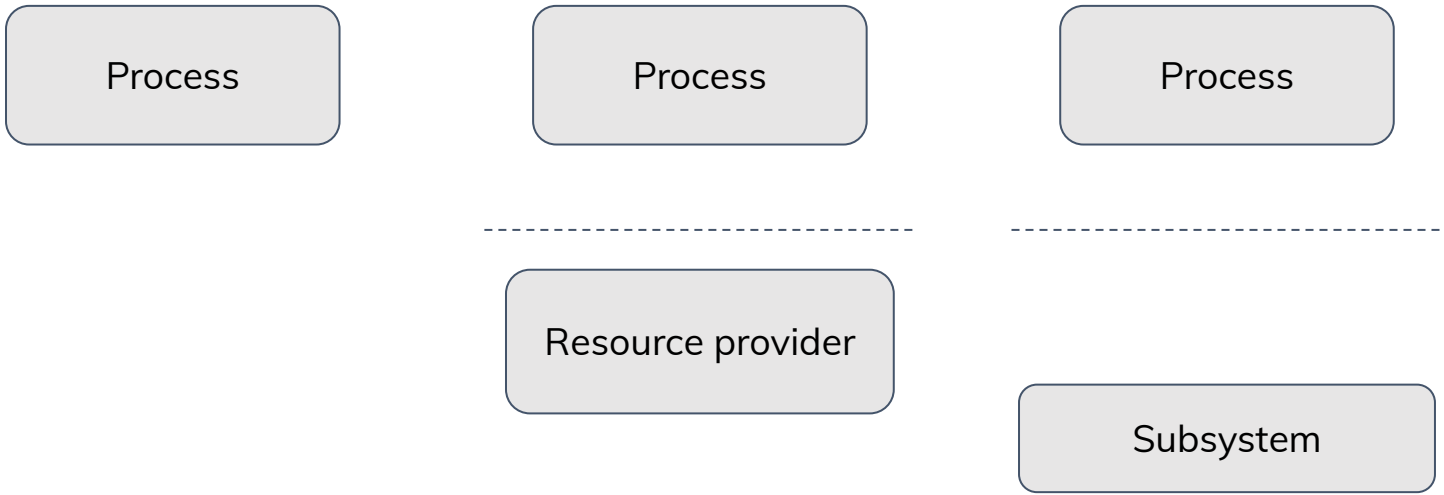
Thought experiment

How do we notify the user-space process about the resource being now gone?

Option A:
Kill the process?

Option B:
Require the process to setup a secondary channel?

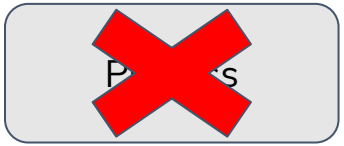
Option C:
Return an error from the next system call?



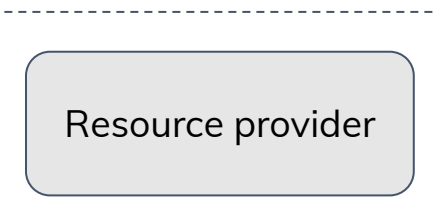
Thought experiment

How do we notify the user-space process about the resource being now gone?

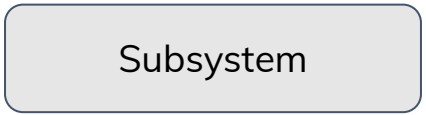
Option A:
Kill the process?



Option B:
Require the process to setup a secondary channel?



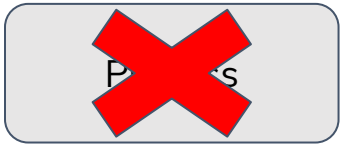
Option C:
Return an error from the next system call?



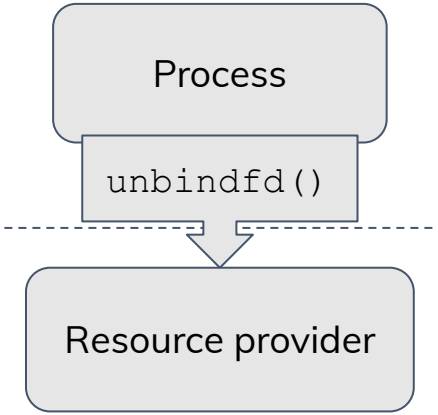
Thought experiment

How do we notify the user-space process about the resource being now gone?

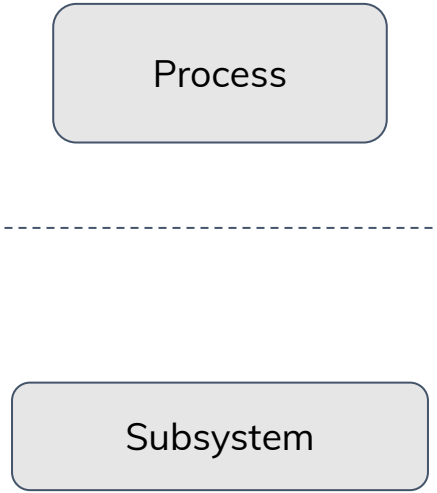
Option A:
Kill the process?



Option B:
Require the process to setup a secondary channel?



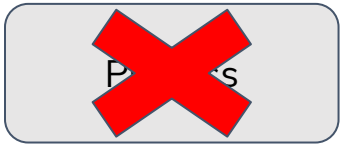
Option C:
Return an error from the next system call?



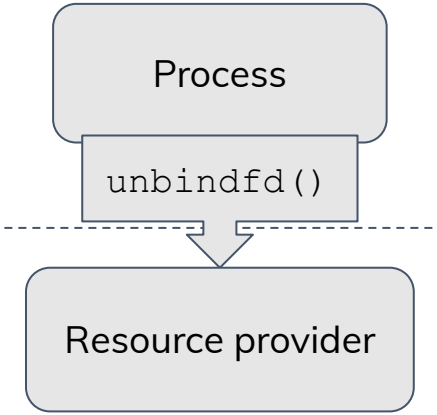
Thought experiment

How do we notify the user-space process about the resource being now gone?

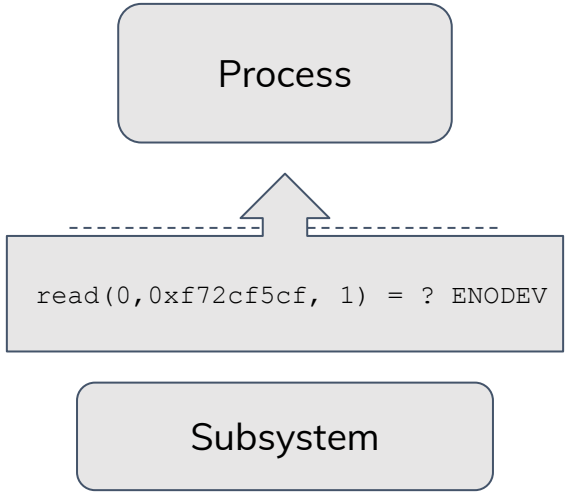
Option A:
Kill the process?



Option B:
Require the process to setup a secondary channel?



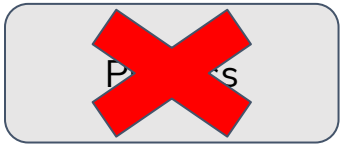
Option C:
Return an error from the next system call?



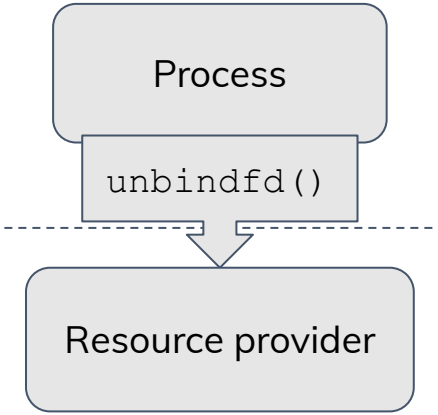
Thought experiment

How do we notify the user-space process about the resource being now gone?

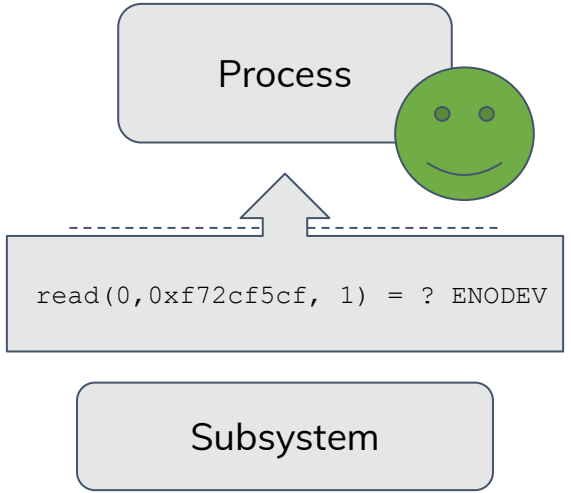
Option A:
Kill the process?



Option B:
Require the process to setup a secondary channel?

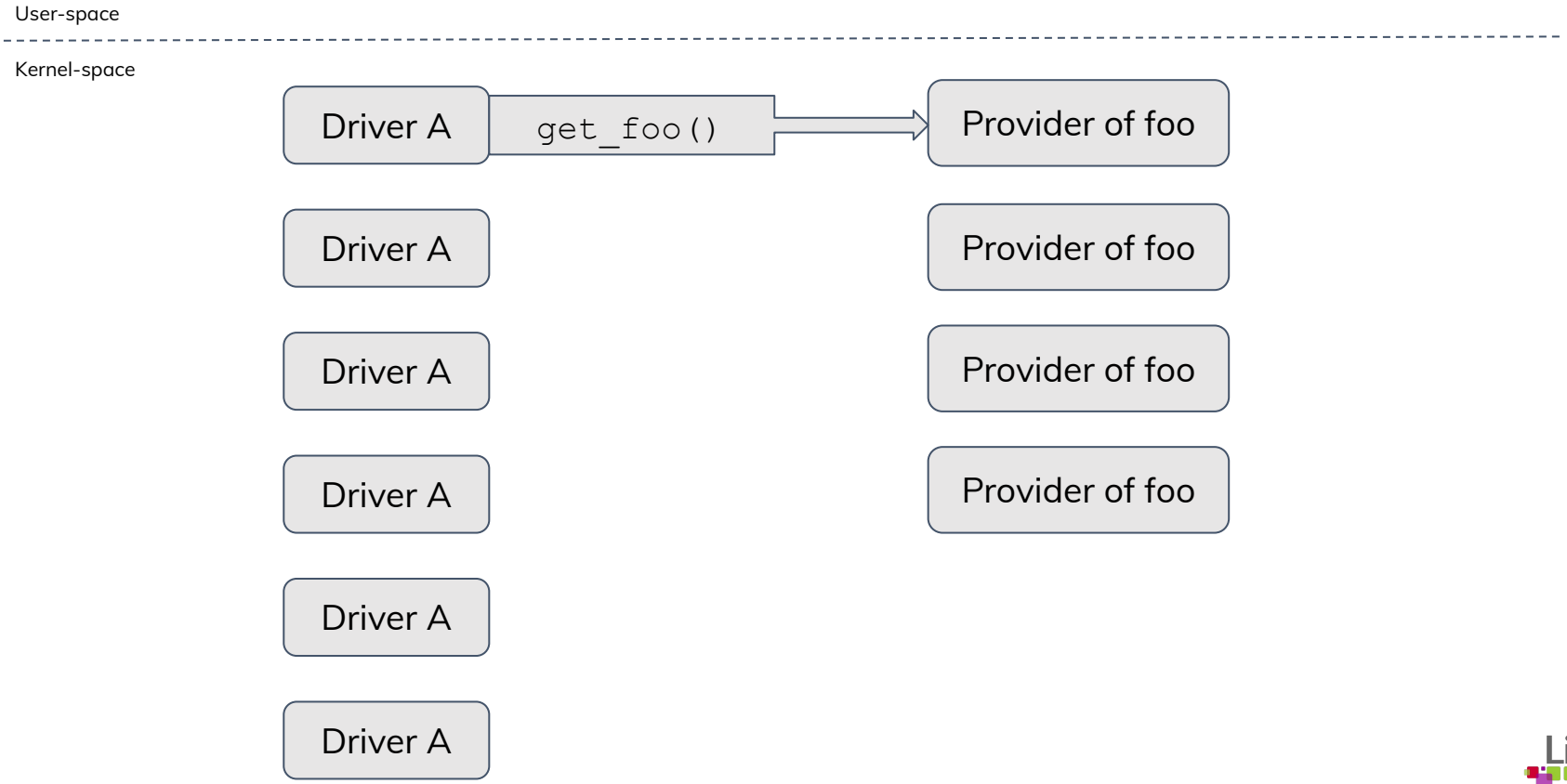


Option C:
Return an error from the next system call?

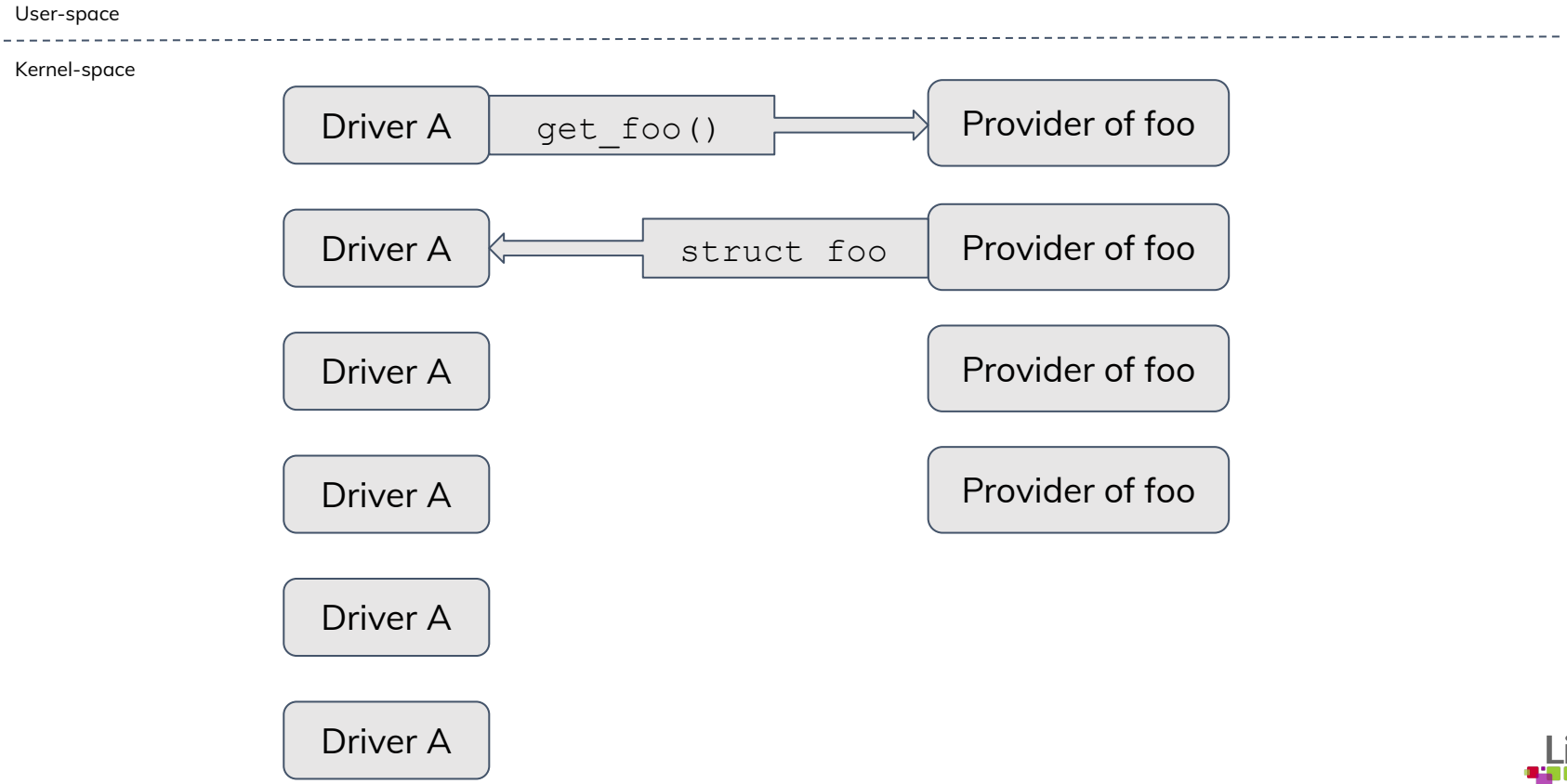


Thought experiment - take 2

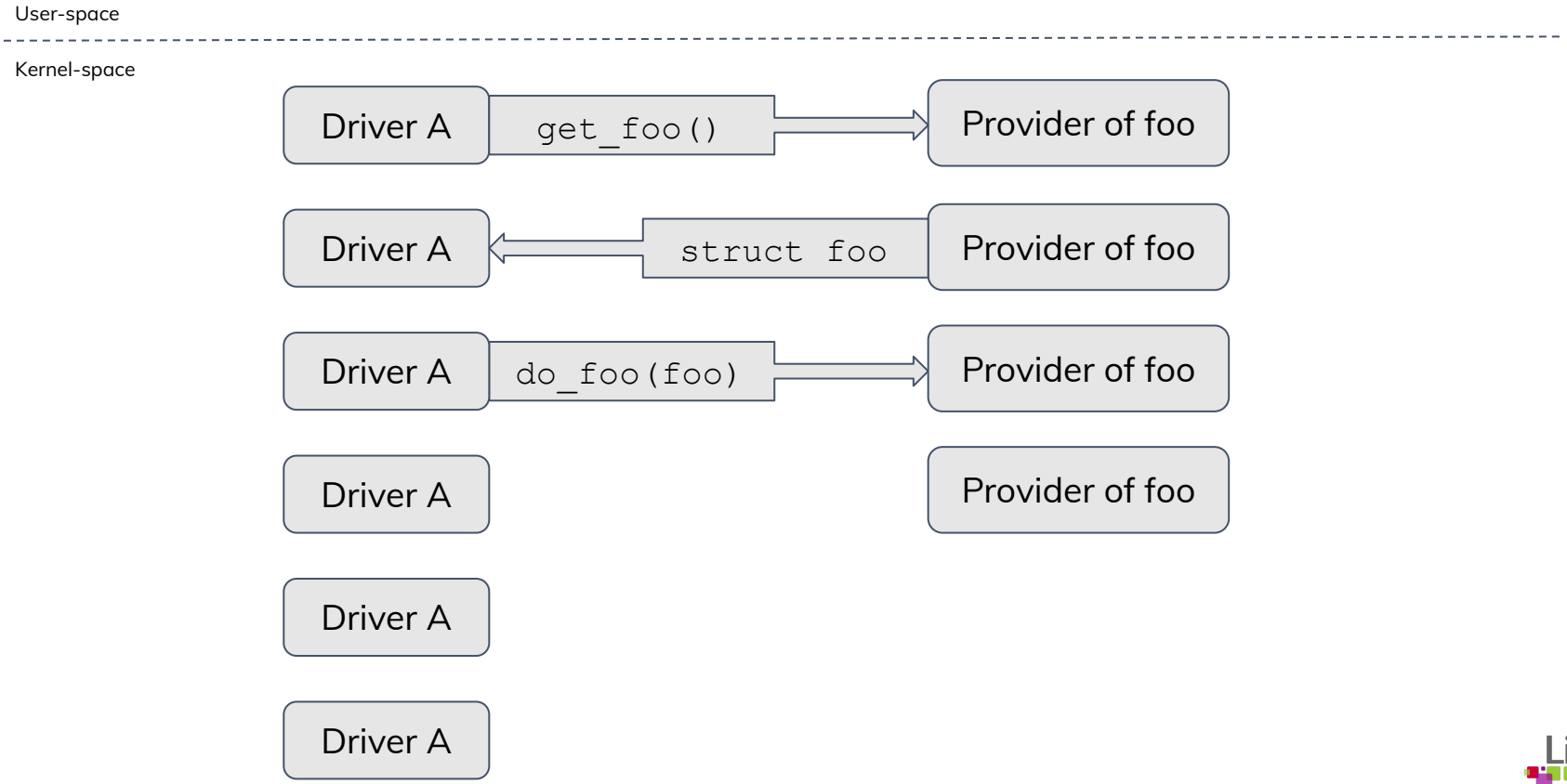
Thought experiment - take 2



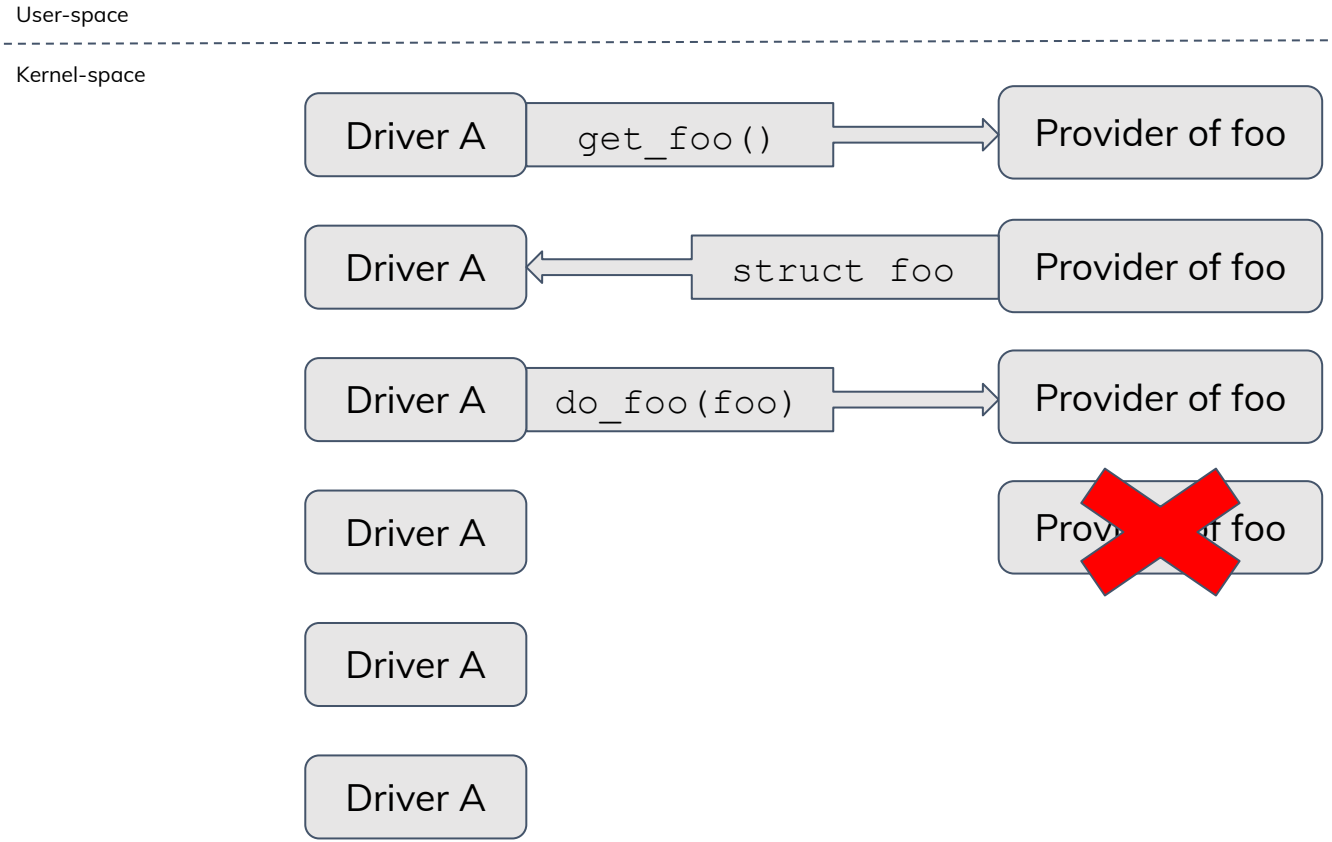
Thought experiment - take 2



Thought experiment - take 2



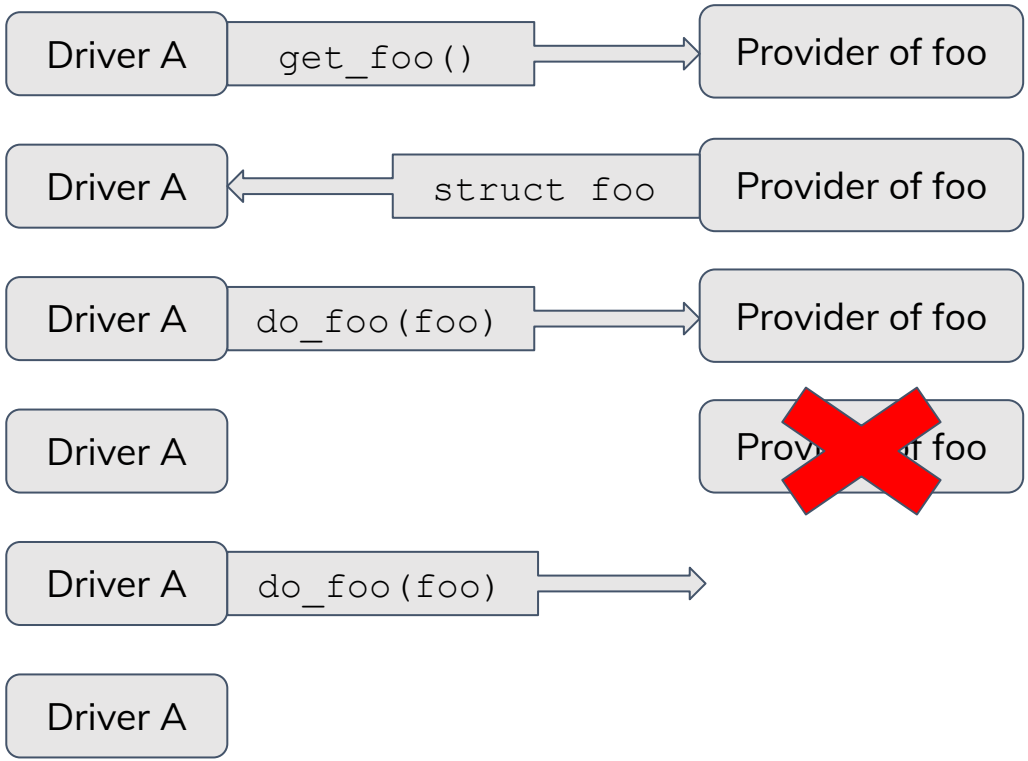
Thought experiment - take 2



Thought experiment - take 2

User-space

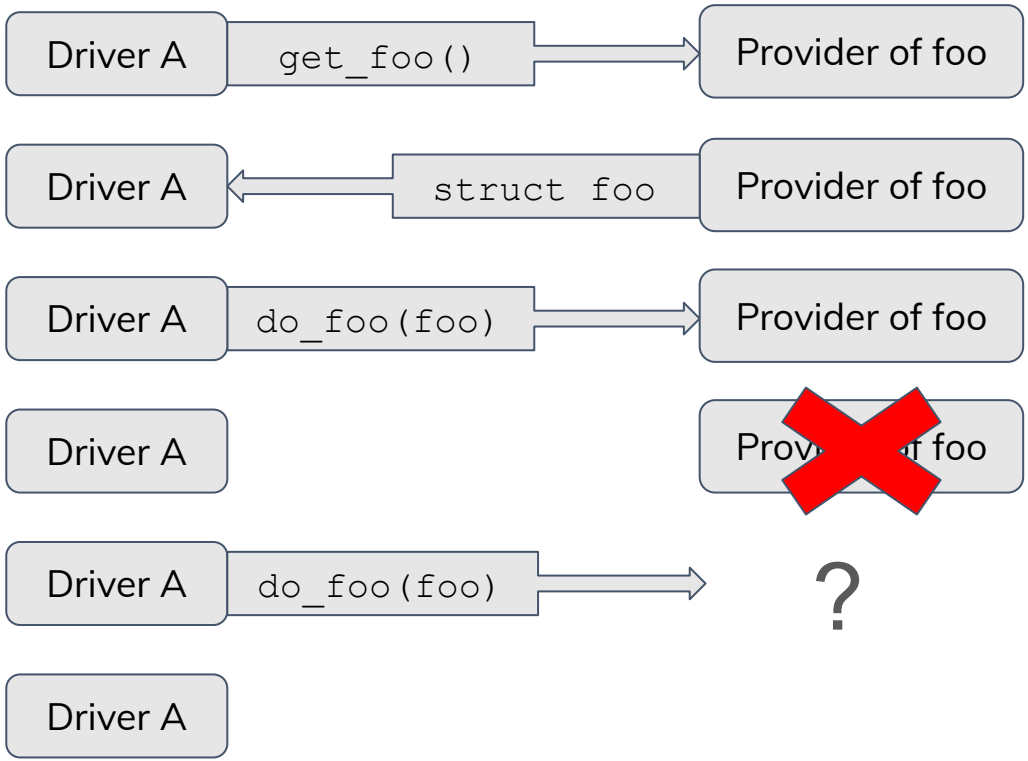
Kernel-space



Thought experiment - take 2

User-space

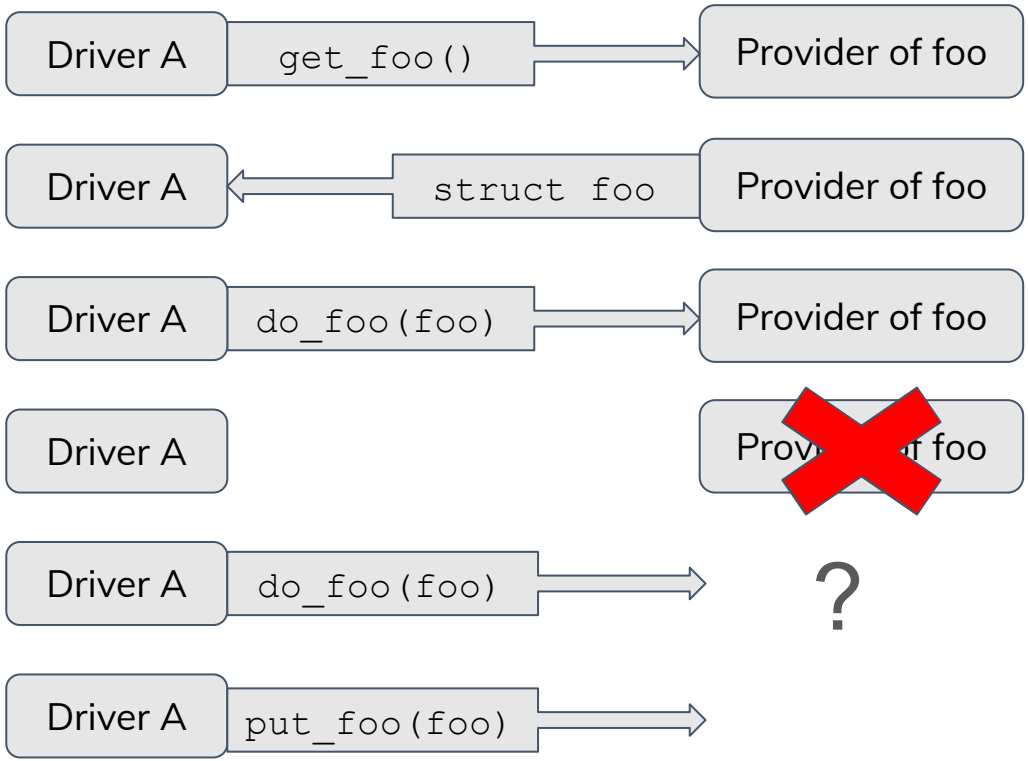
Kernel-space



Thought experiment - take 2

User-space

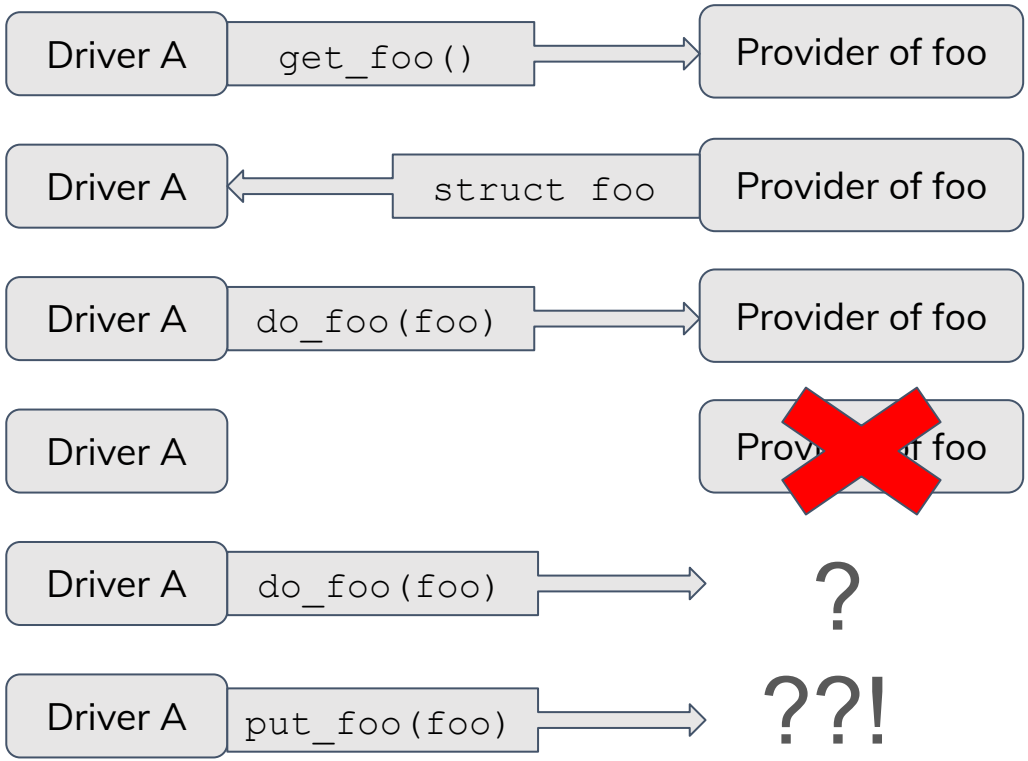
Kernel-space



Thought experiment - take 2

User-space

Kernel-space

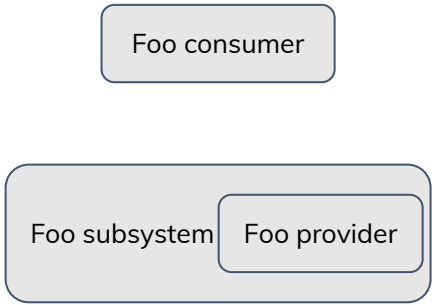


Thought experiment - take 2

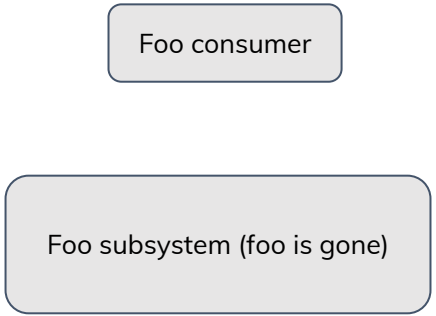
How do we notify the driver about the resource being now gone?

Option A:
Force device unbind?

Option B:
Require the driver to setup a secondary channel?



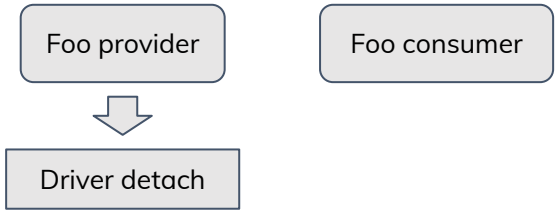
Option C:
Return an error from the next API call?



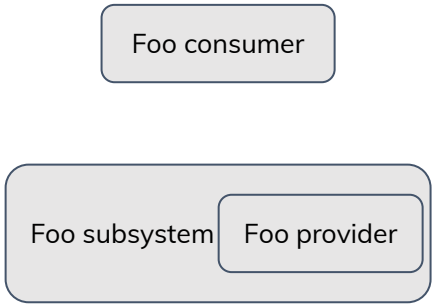
Thought experiment - take 2

How do we notify the driver about the resource being now gone?

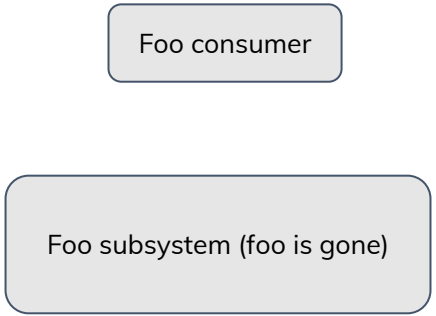
Option A:
Force device unbind?



Option B:
Require the driver to setup a secondary channel?



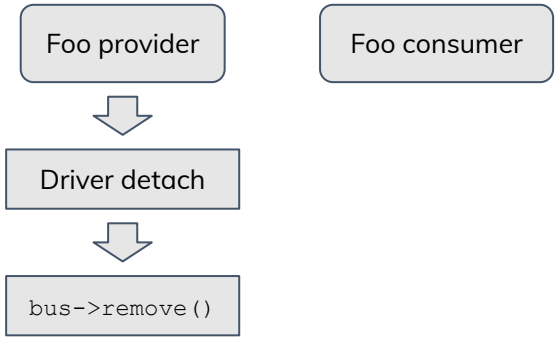
Option C:
Return an error from the next API call?



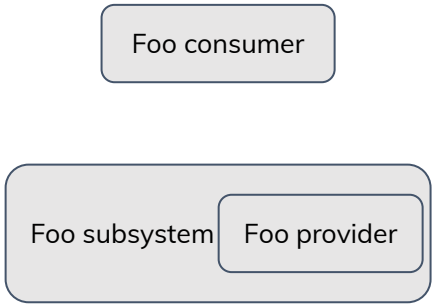
Thought experiment - take 2

How do we notify the driver about the resource being now gone?

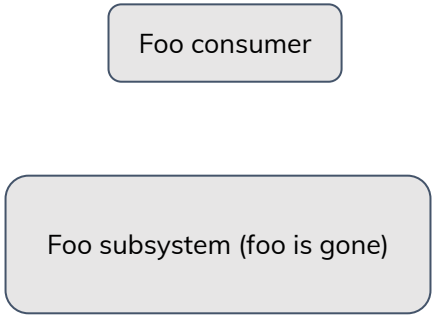
Option A:
Force device unbind?



Option B:
Require the driver to setup a secondary channel?



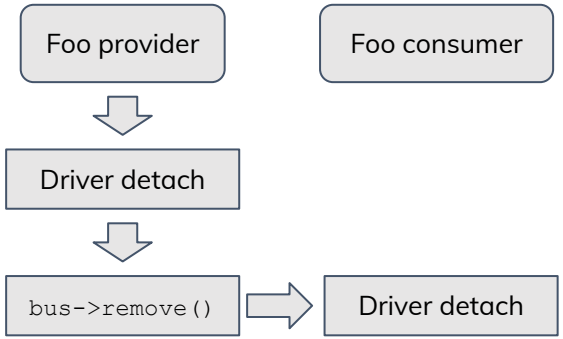
Option C:
Return an error from the next API call?



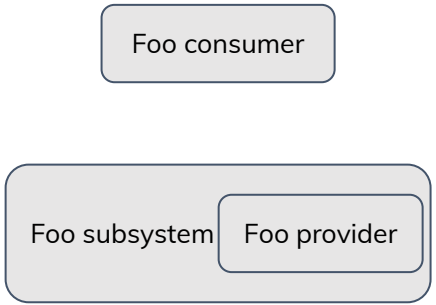
Thought experiment - take 2

How do we notify the driver about the resource being now gone?

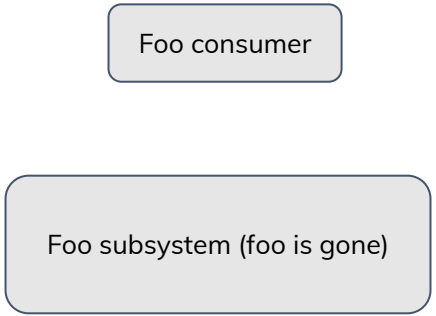
Option A:
Force device unbind?



Option B:
Require the driver to setup a secondary channel?



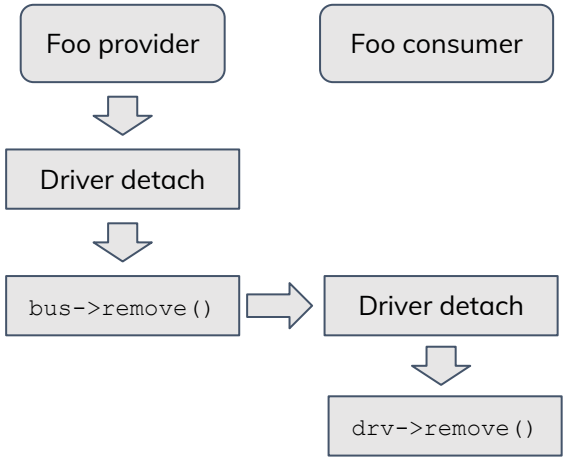
Option C:
Return an error from the next API call?



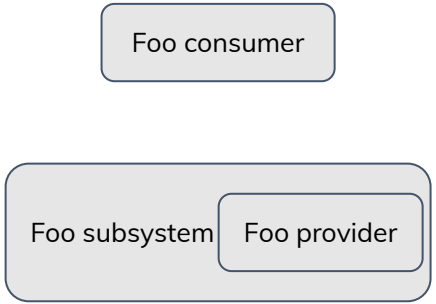
Thought experiment - take 2

How do we notify the driver about the resource being now gone?

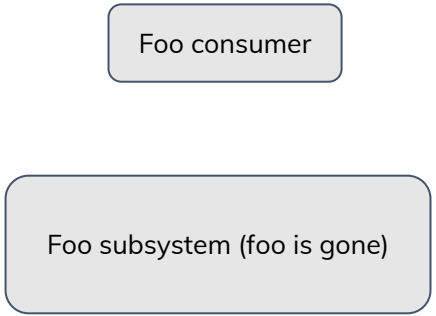
Option A:
Force device unbind?



Option B:
Require the driver to setup a secondary channel?



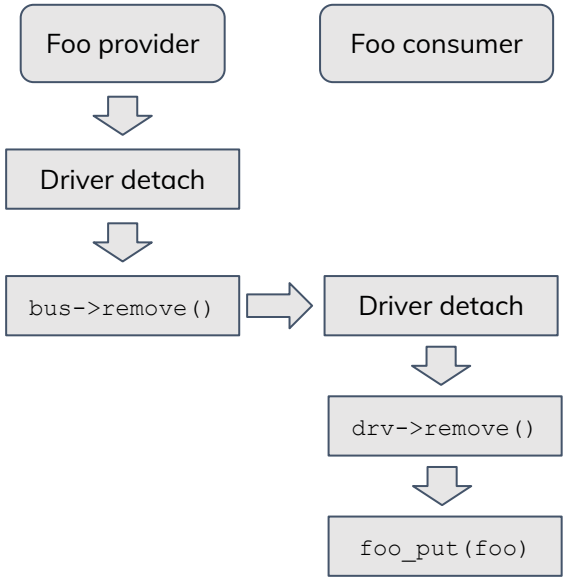
Option C:
Return an error from the next API call?



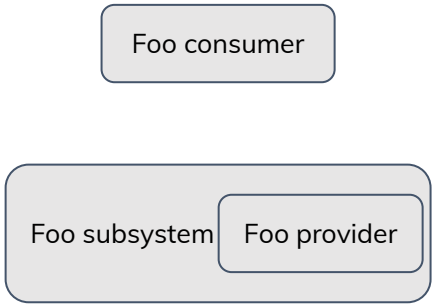
Thought experiment - take 2

How do we notify the driver about the resource being now gone?

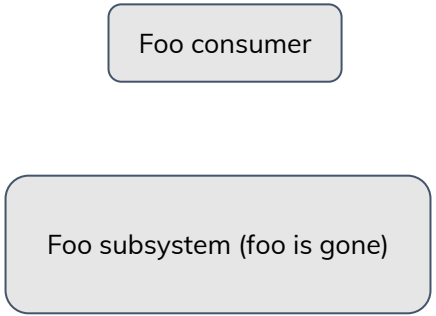
Option A:
Force device unbind?



Option B:
Require the driver to setup a secondary channel?



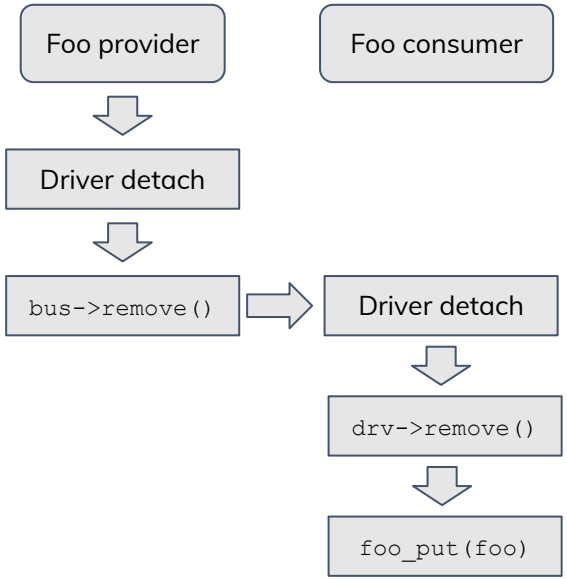
Option C:
Return an error from the next API call?



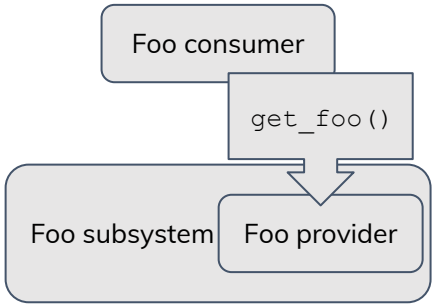
Thought experiment - take 2

How do we notify the driver about the resource being now gone?

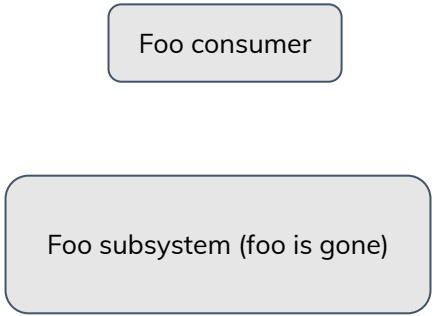
Option A:
Force device unbind?



Option B:
Require the driver to setup a secondary channel?



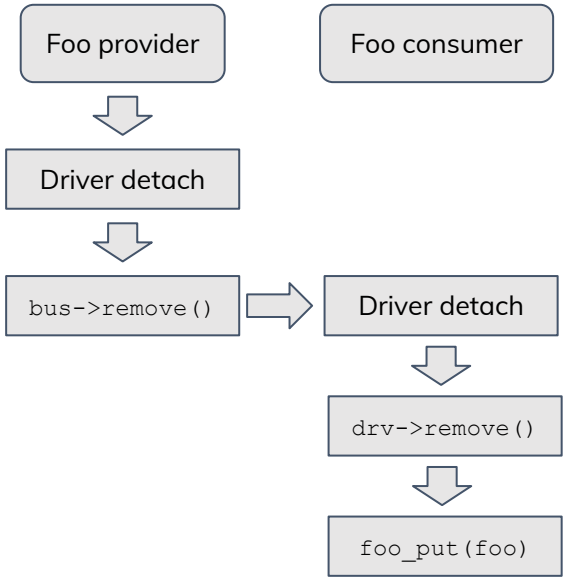
Option C:
Return an error from the next API call?



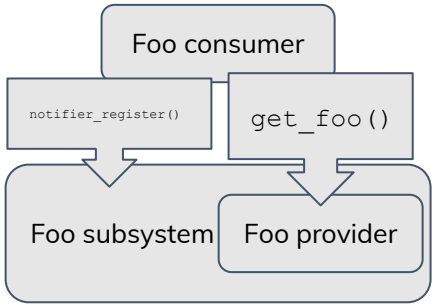
Thought experiment - take 2

How do we notify the driver about the resource being now gone?

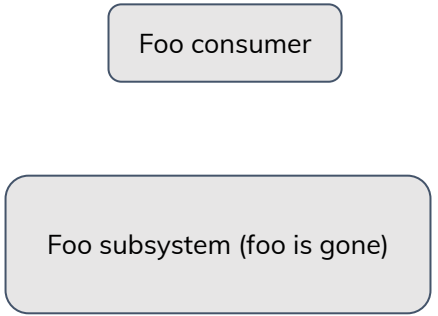
Option A:
Force device unbind?



Option B:
Require the driver to setup a secondary channel?



Option C:
Return an error from the next API call?

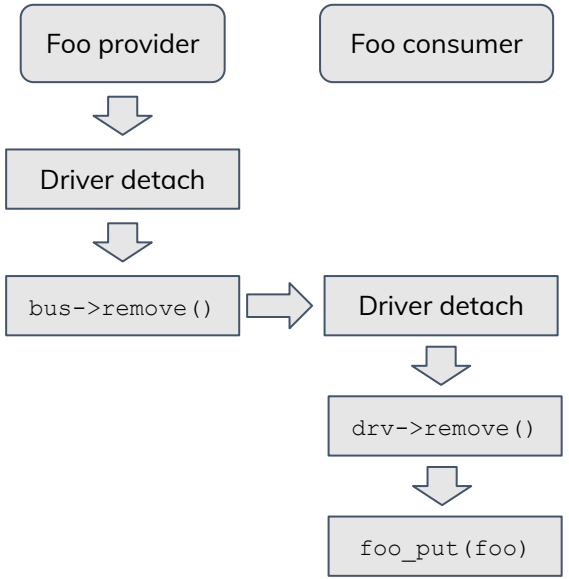


Linaro

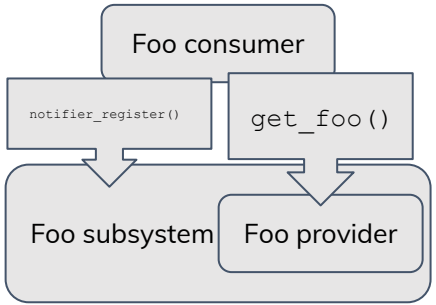
Thought experiment - take 2

How do we notify the driver about the resource being now gone?

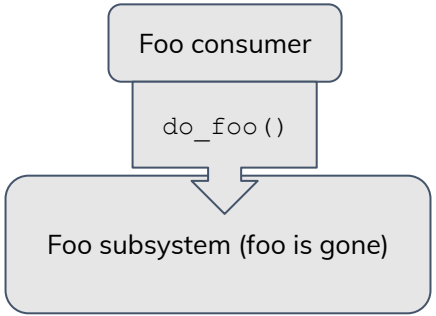
Option A:
Force device unbind?



Option B:
Require the driver to setup a secondary channel?



Option C:
Return an error from the next API call?

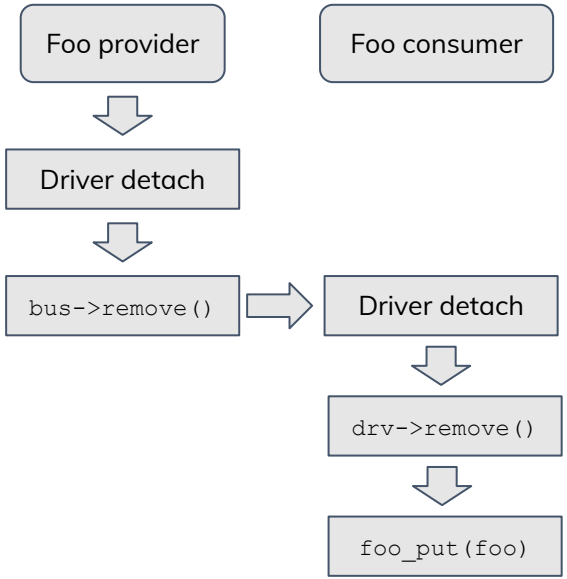


Linaro

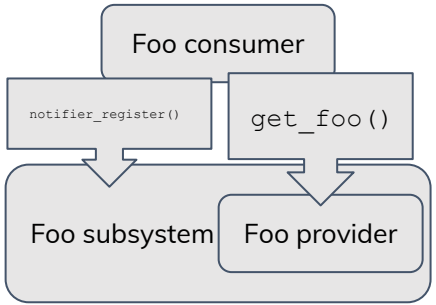
Thought experiment - take 2

How do we notify the driver about the resource being now gone?

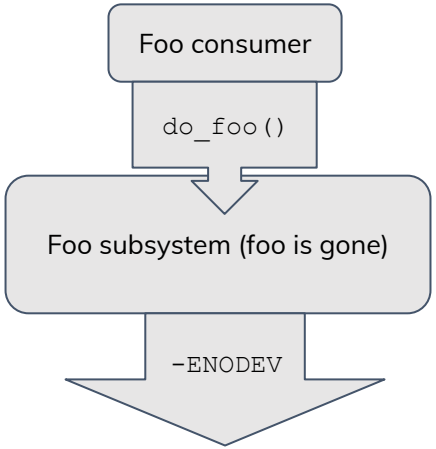
Option A:
Force device unbind?



Option B:
Require the driver to setup a secondary channel?



Option C:
Return an error from the next API call?



Linaro

Real-life example

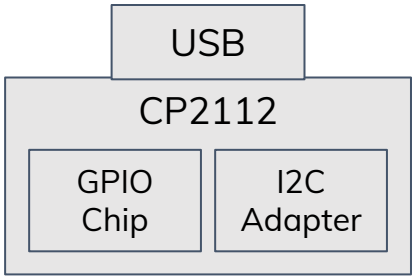
User-space

Kernel-space

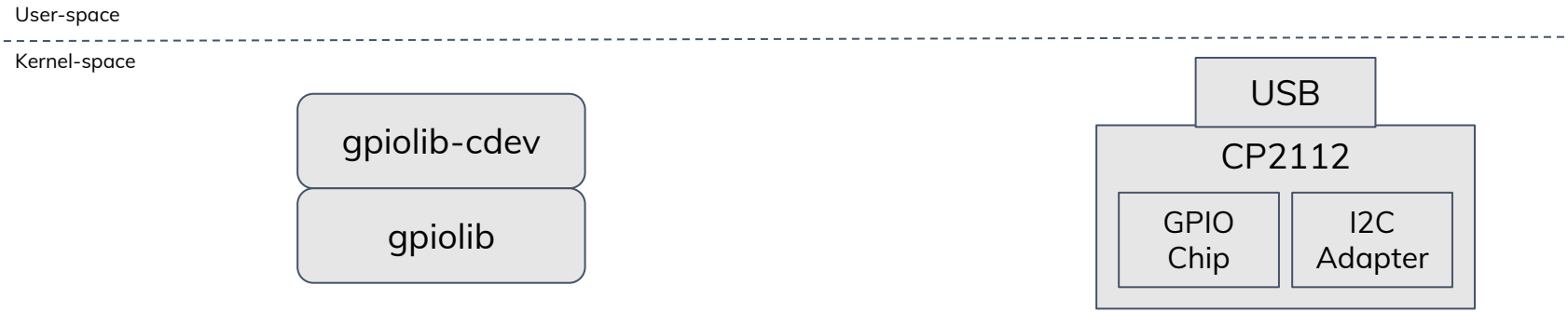
Real-life example

User-space

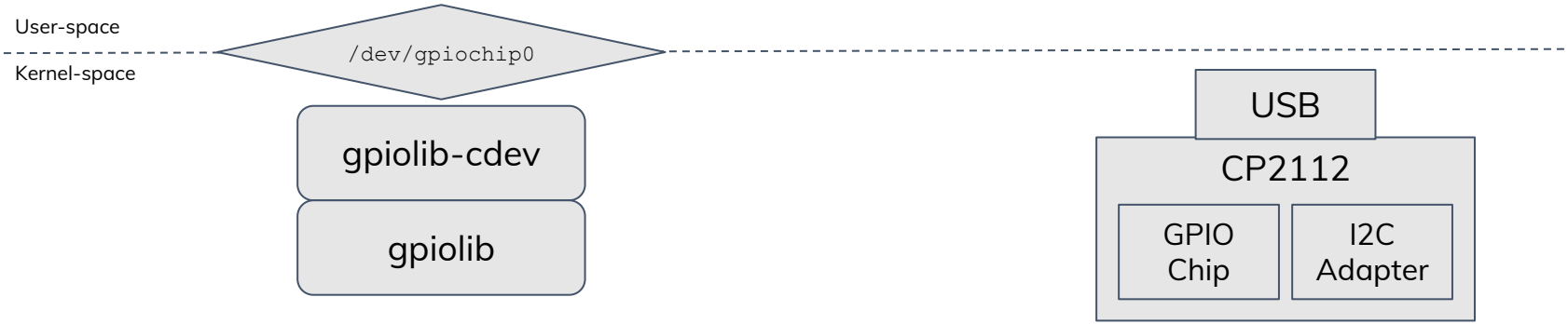
Kernel-space



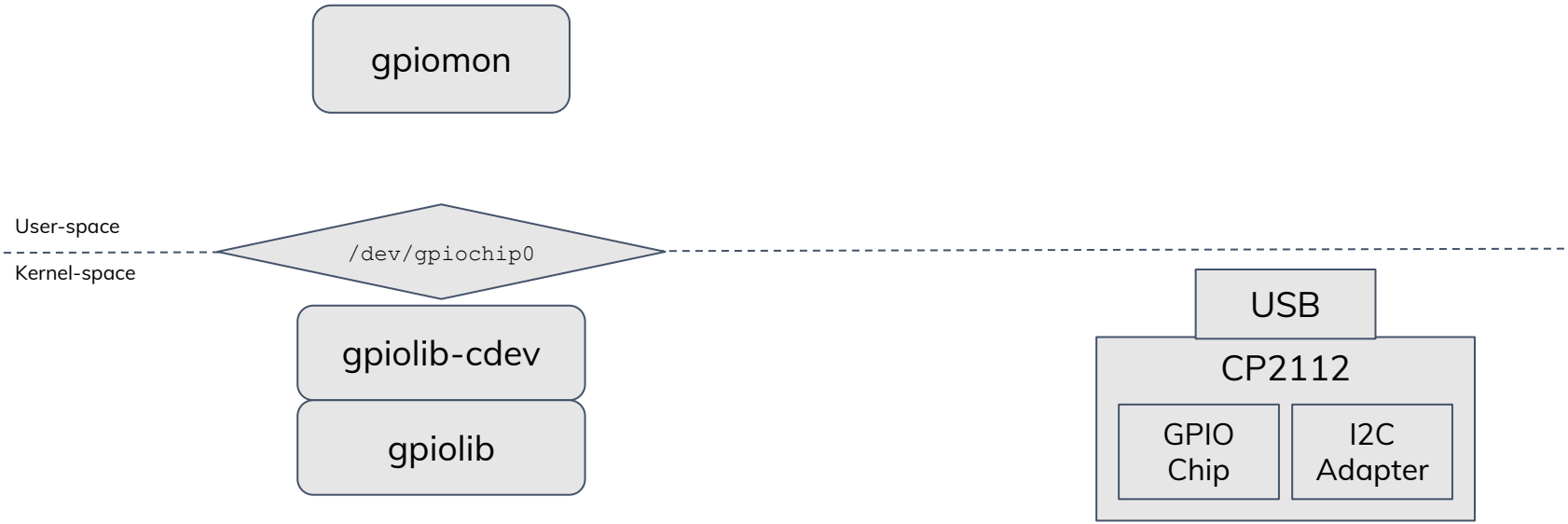
Real-life example



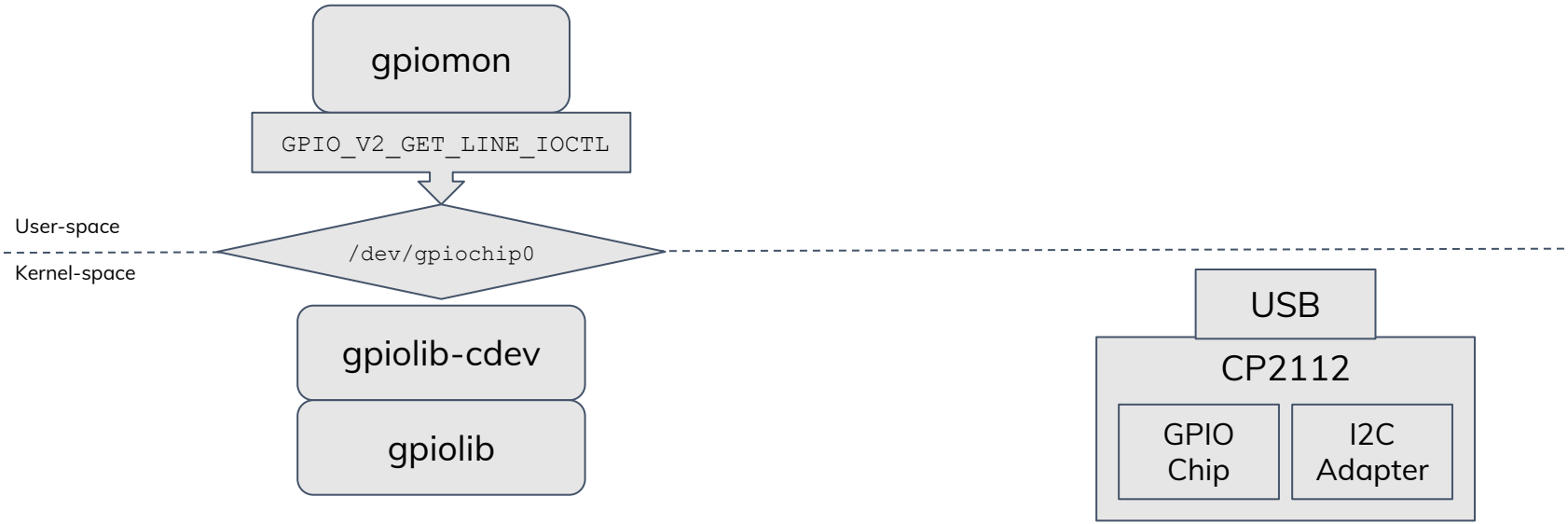
Real-life example



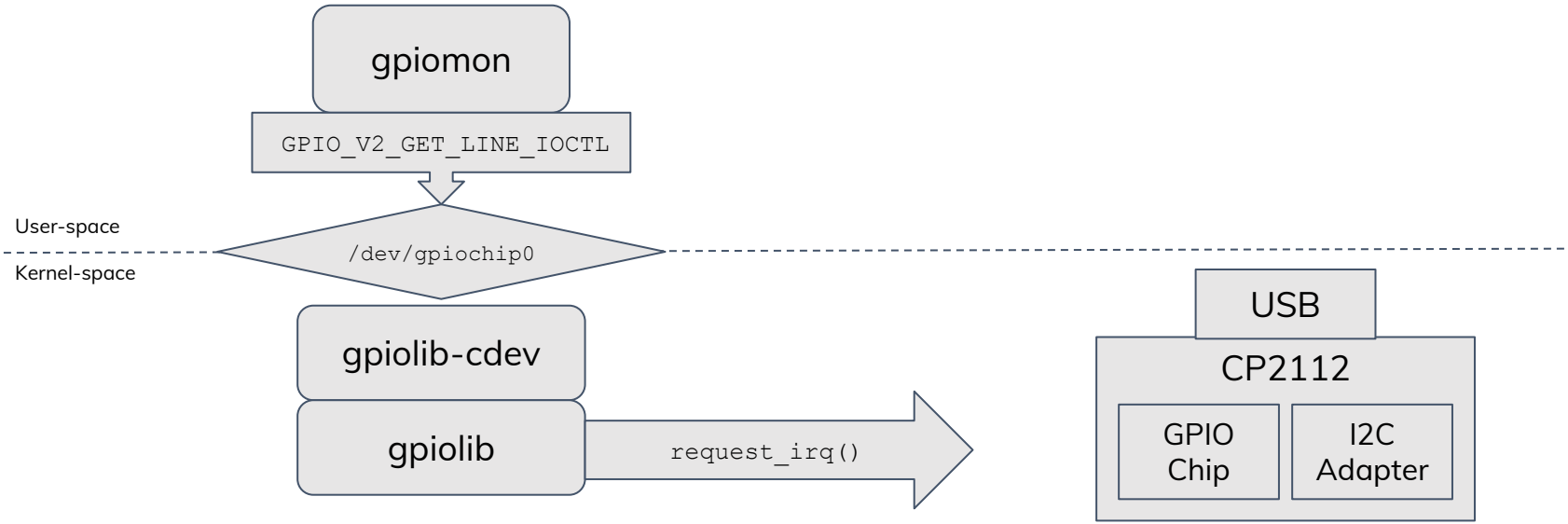
Real-life example



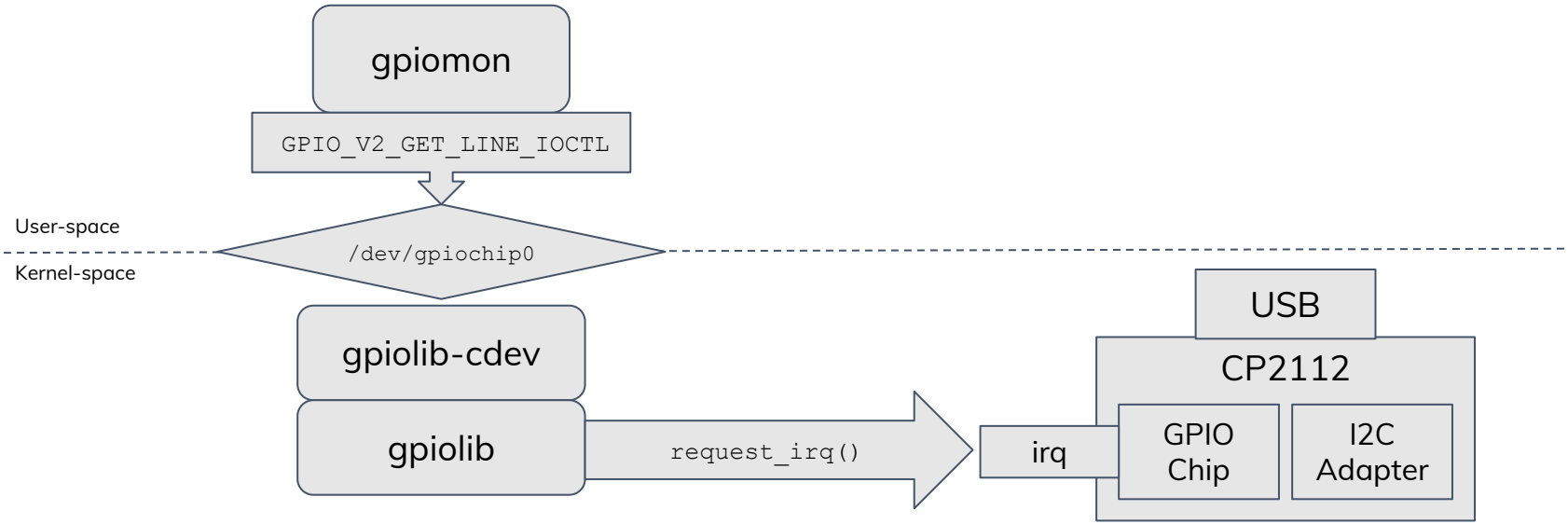
Real-life example



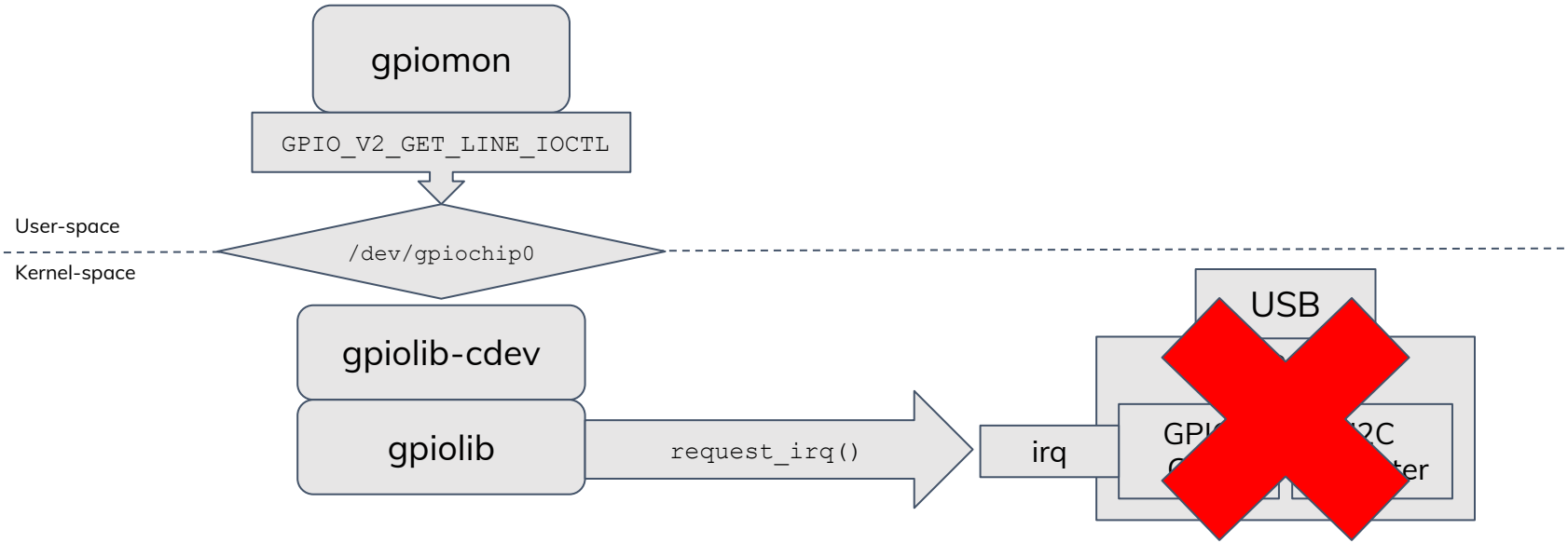
Real-life example



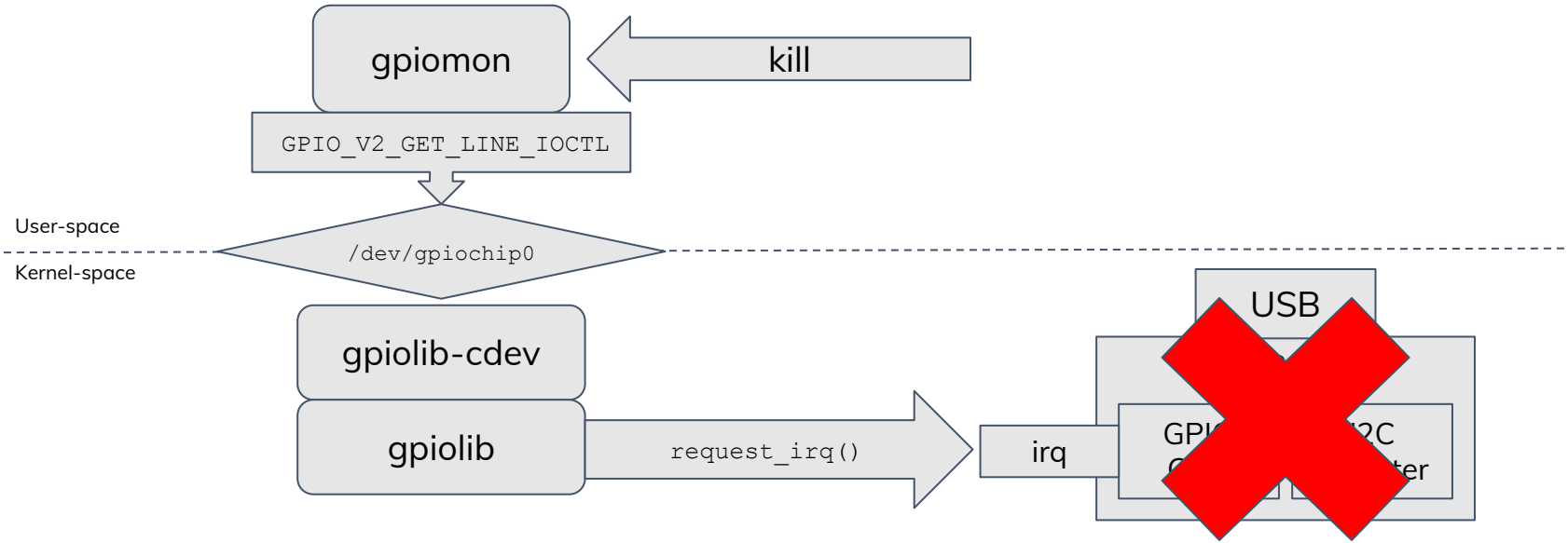
Real-life example



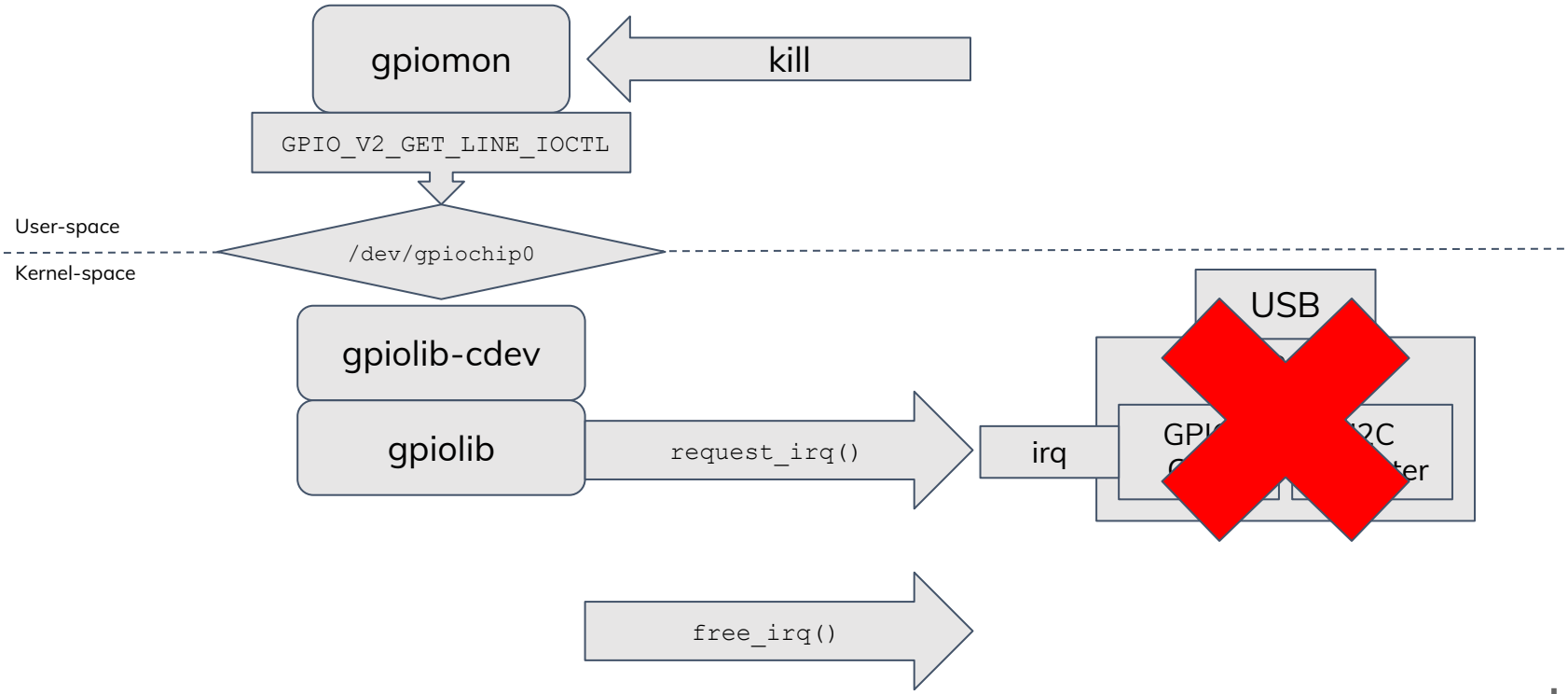
Real-life example



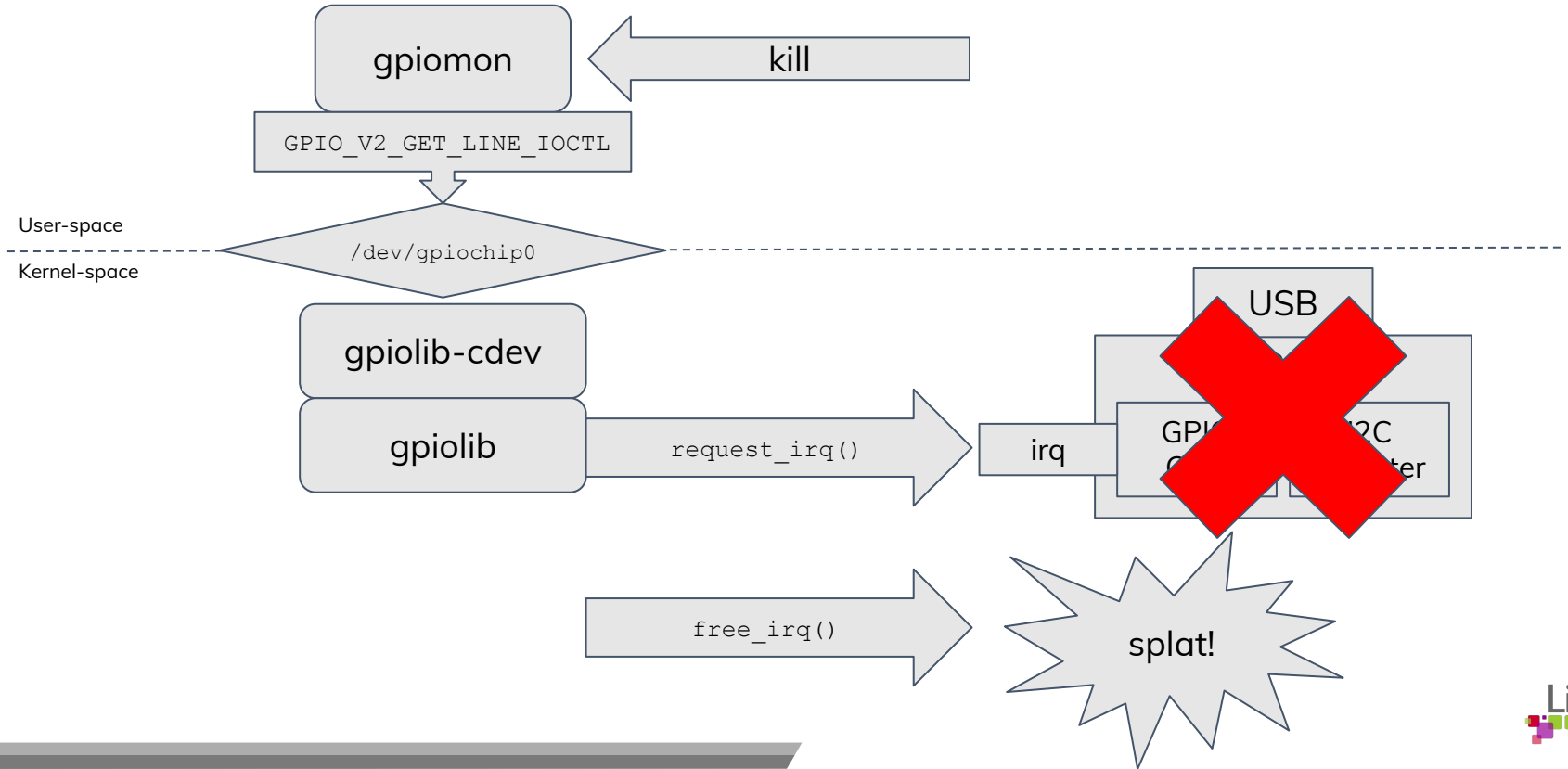
Real-life example



Real-life example



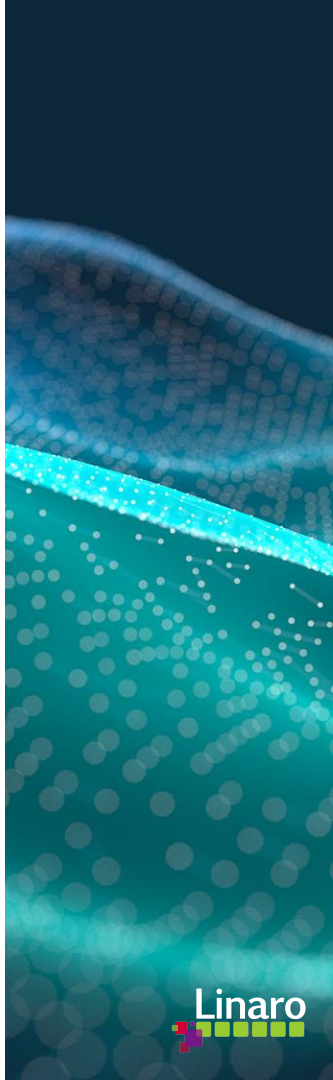
Real-life example



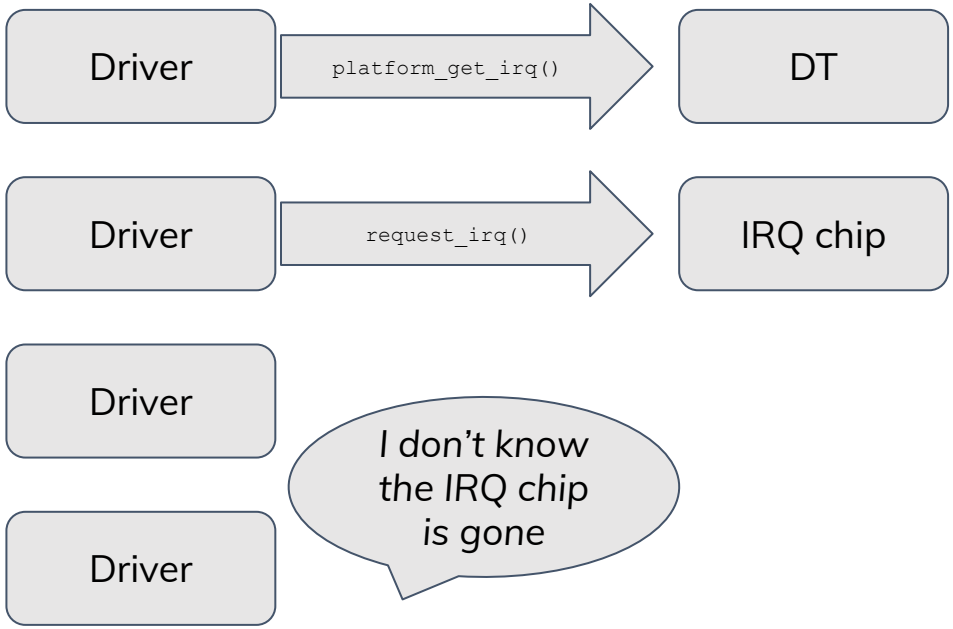
Linaro

Real-life example

- But you could hack something up to notify the character device code I hear you say!

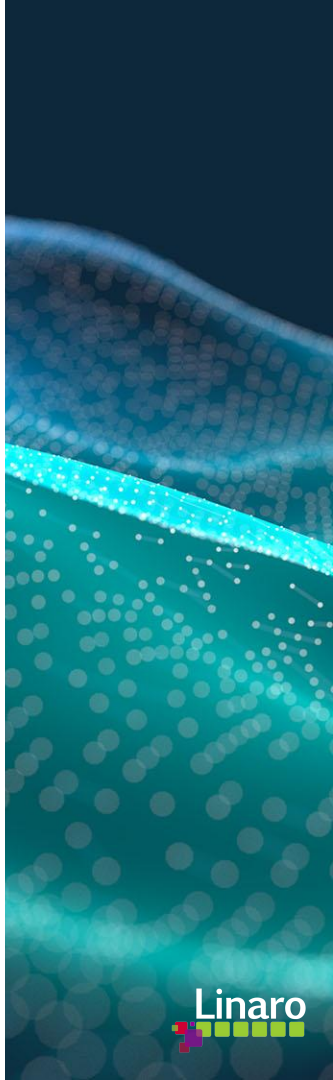


This is generic problem



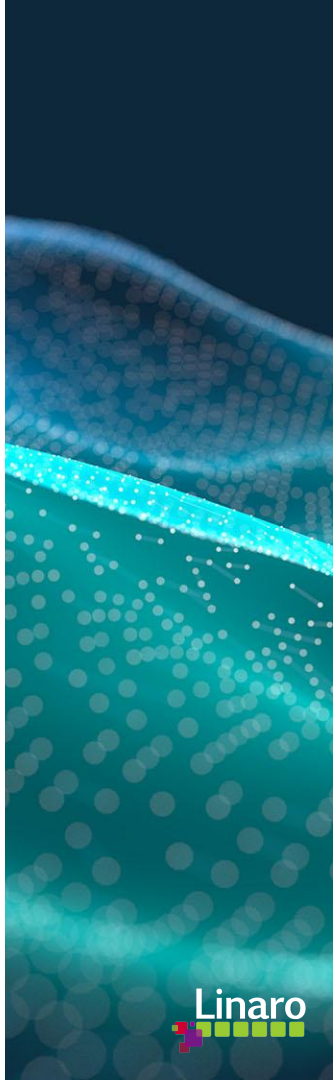
This is generic problem

- Not just interrupts, the same would happen with many other providers



What’s going on?

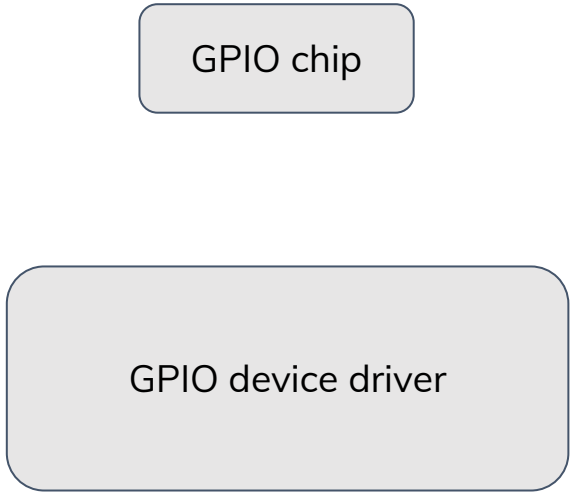
- Consumers of resources are not notified about the providers of these resources getting unbound.
- Consumers hold handles (pointer or number) but there’s no notification mechanism to let them know the provider is gone.
- GPIO can just barely handle it, interrupts cannot at all.



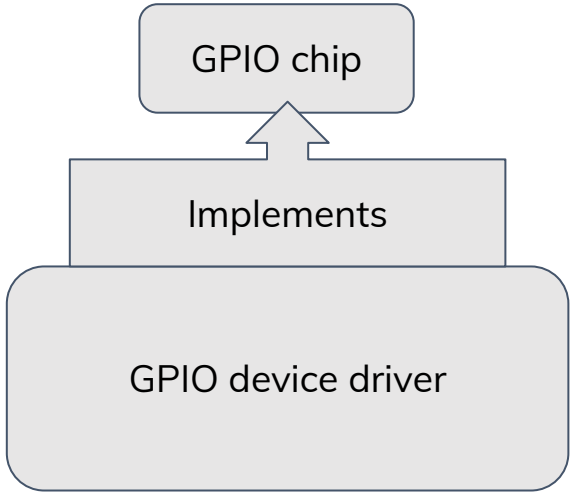
What’s going on?

What’s going on?

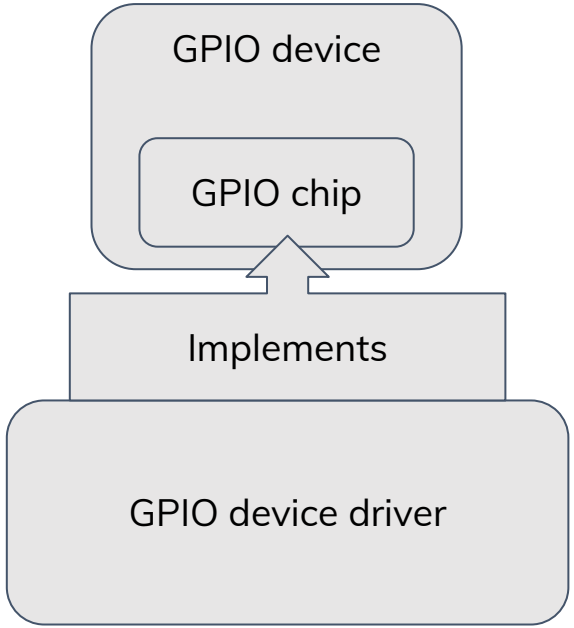
What’s going on?



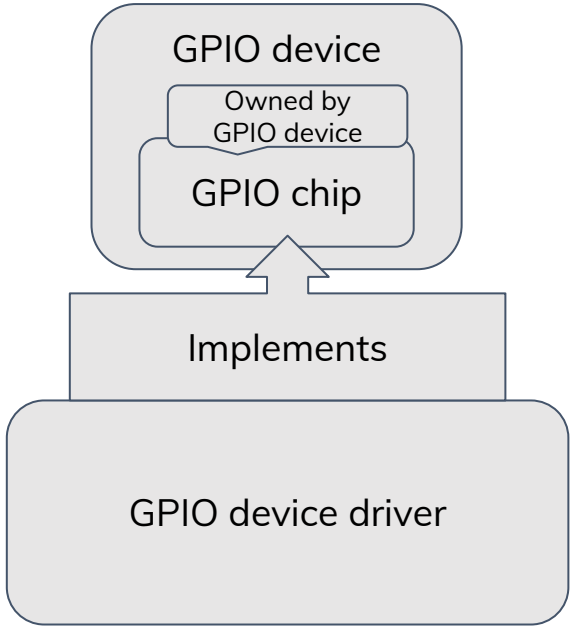
What’s going on?



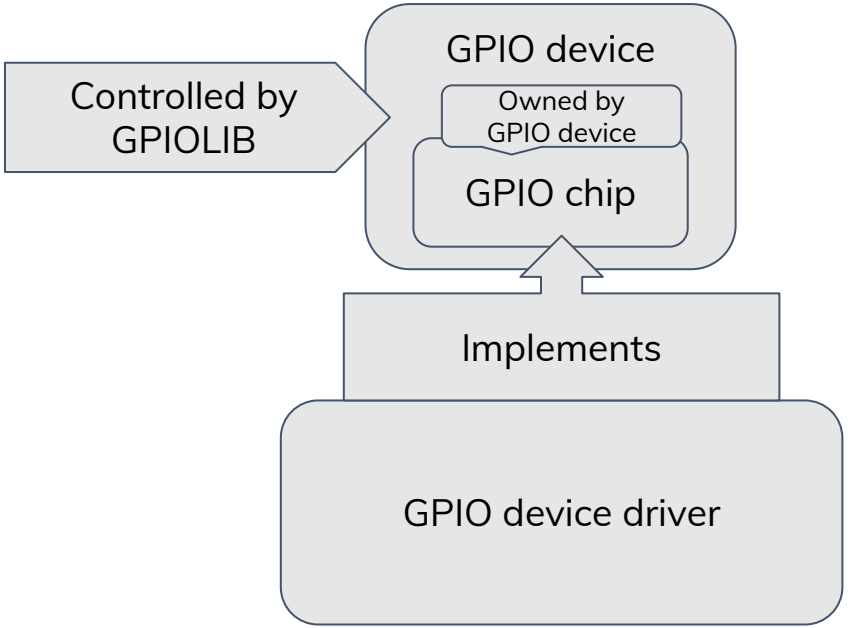
What's going on?



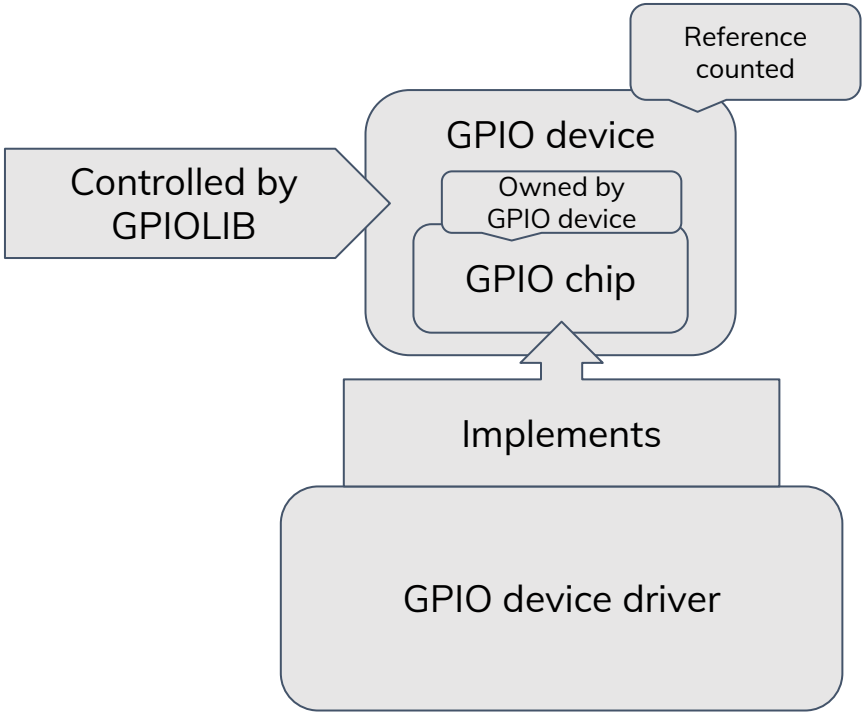
What's going on?



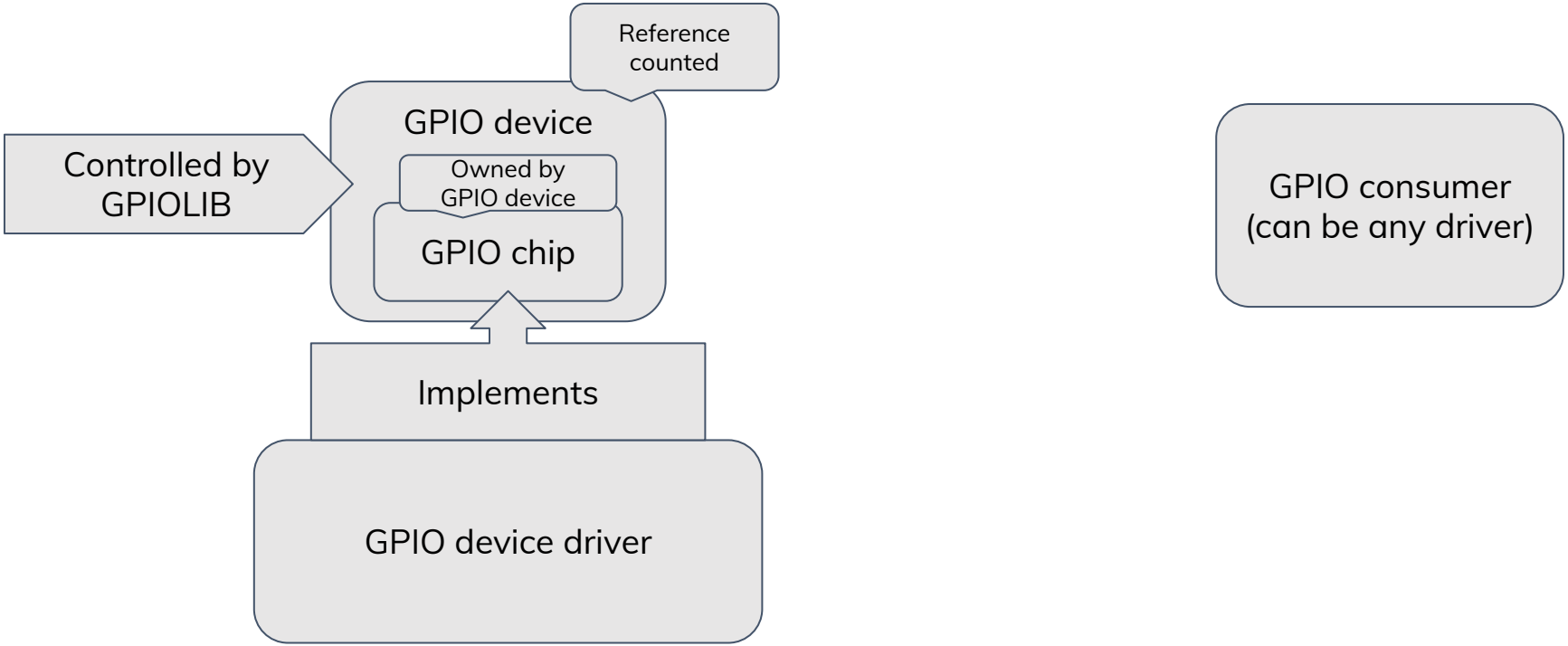
What's going on?



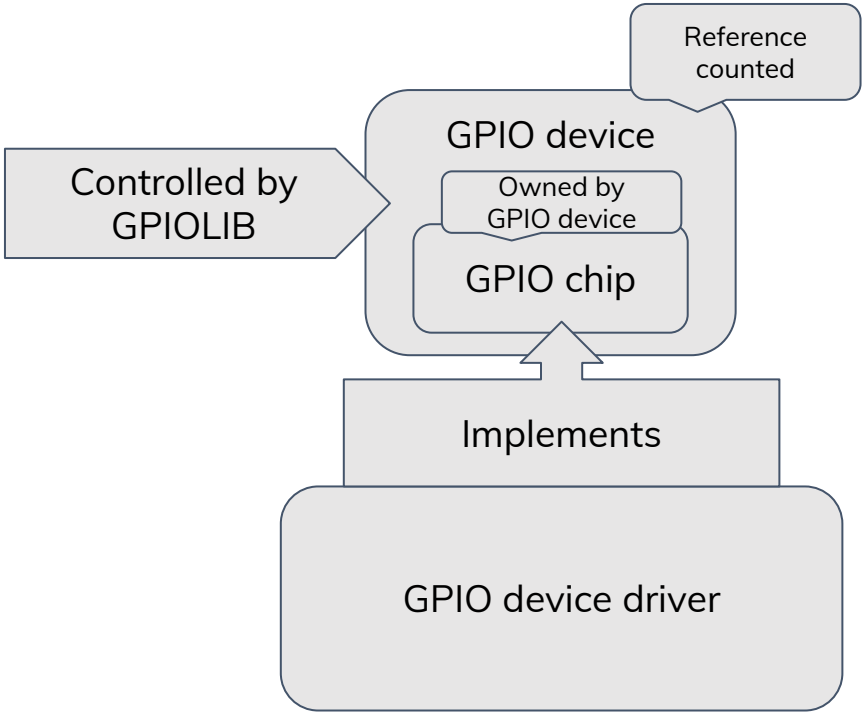
What's going on?



What's going on?



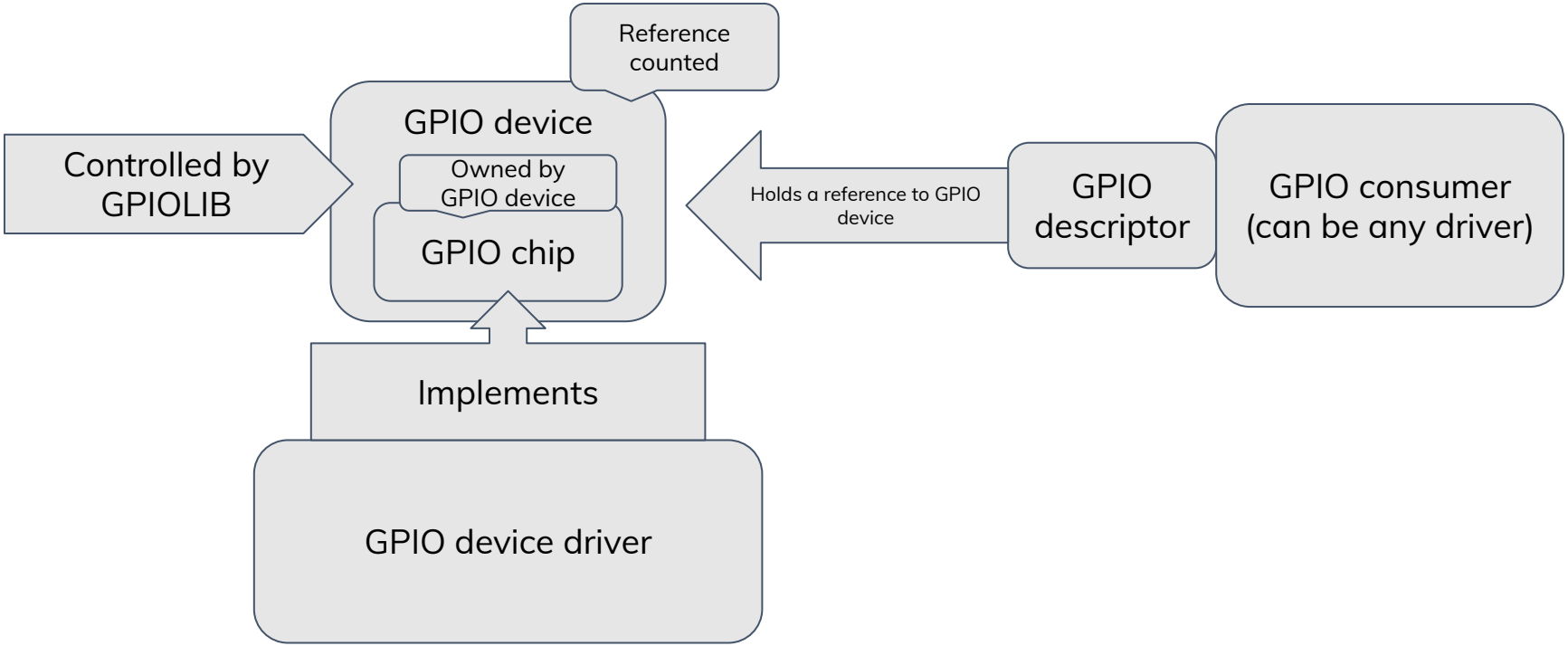
What's going on?



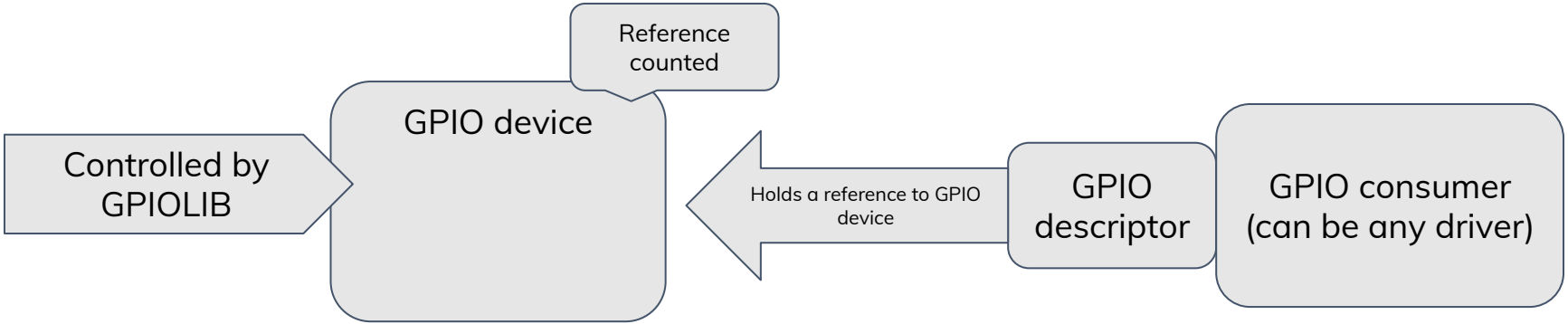
GPIO descriptor

GPIO consumer (can be any driver)

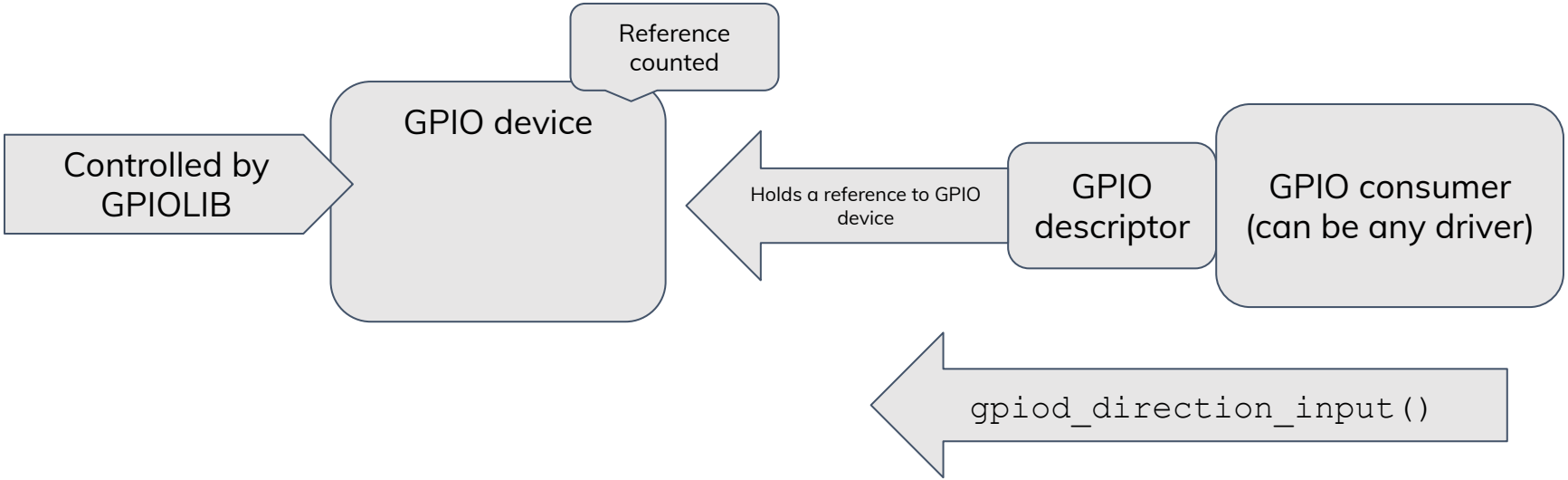
What's going on?



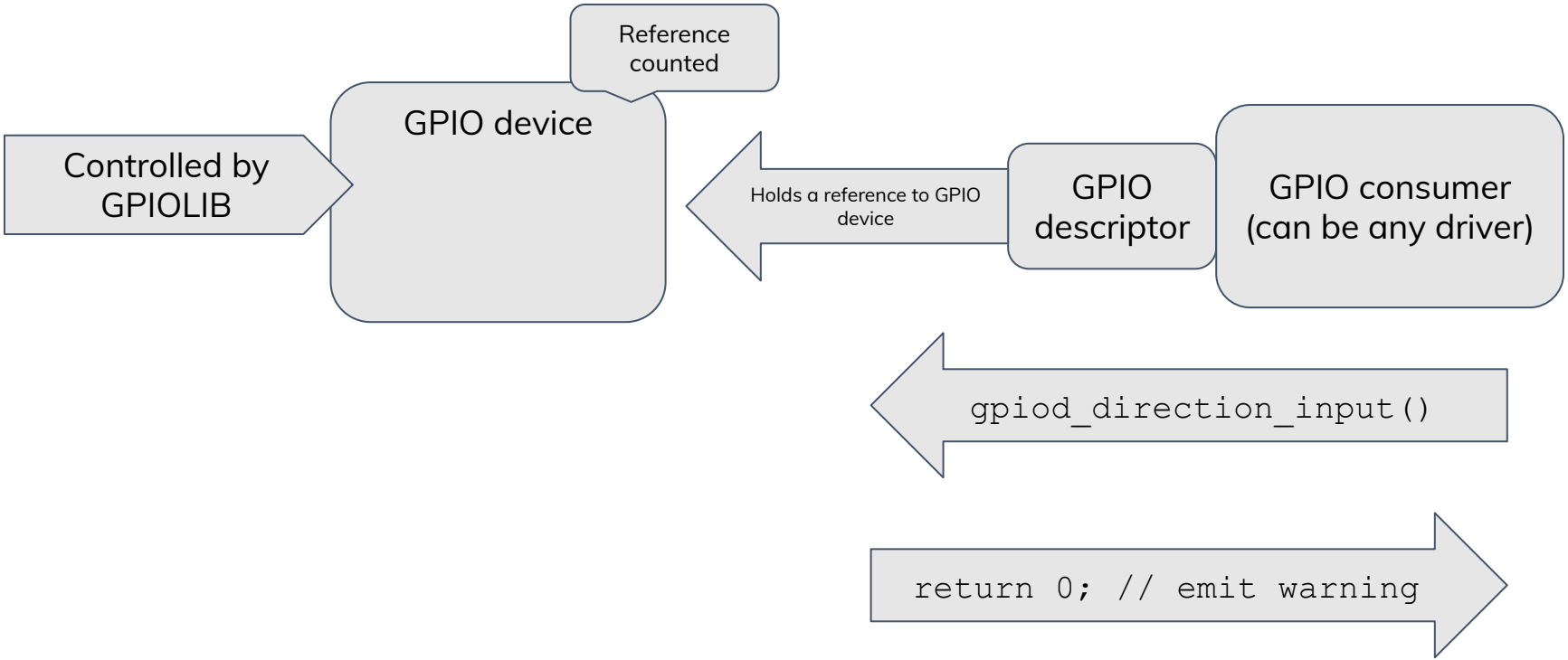
What's going on?



What's going on?

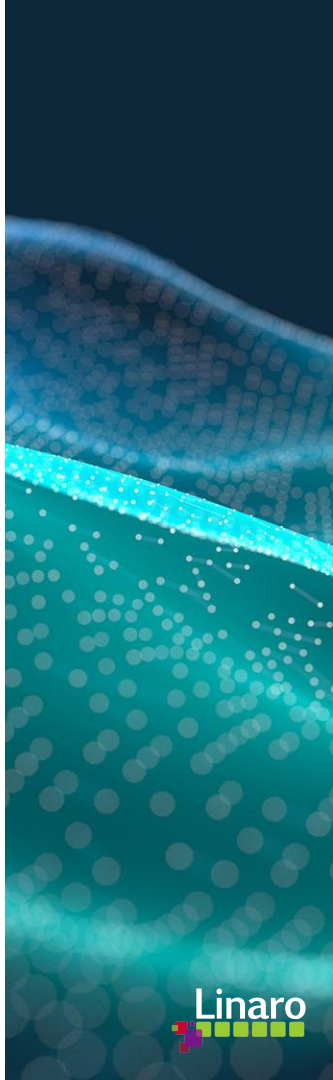


What's going on?



What’s going on?

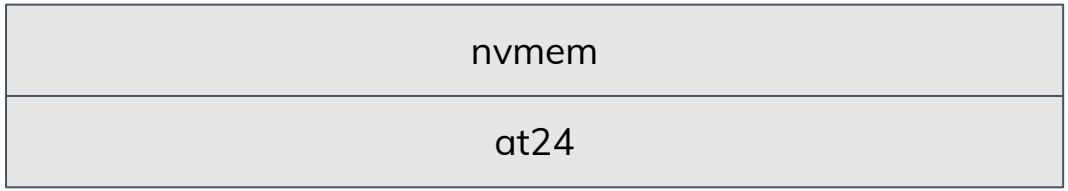
- Yes, GPIOLIB still needs a lot of work for correct plug-and-play
  - Needs fine-grained locking
  - RCU protection of struct gpio_chip pointer
  - Must not use the global spinlock as it releases it at times to interact with pinctrl which uses mutexes exclusively (sic!)



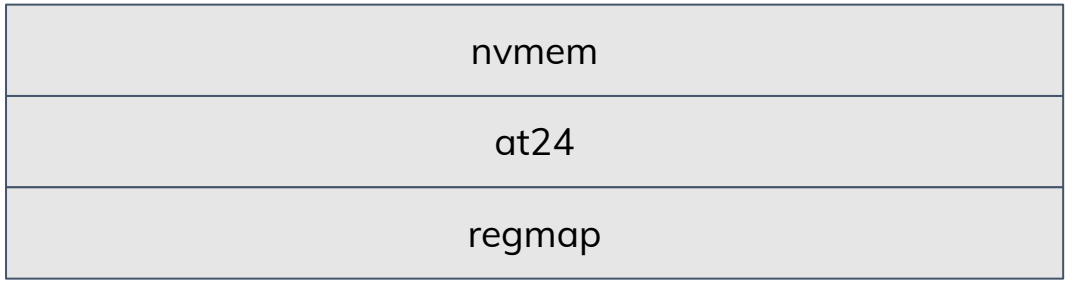
We go through so many abstraction layers...

We go through so many abstraction layers...

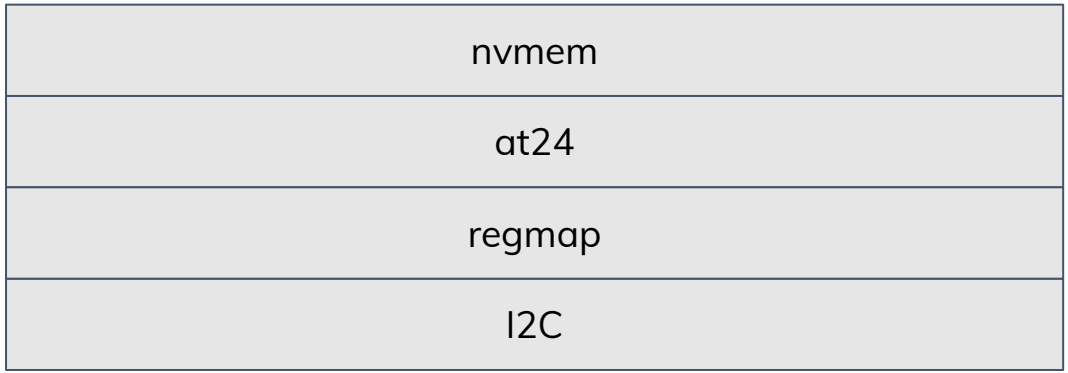
We go through so many abstraction layers...



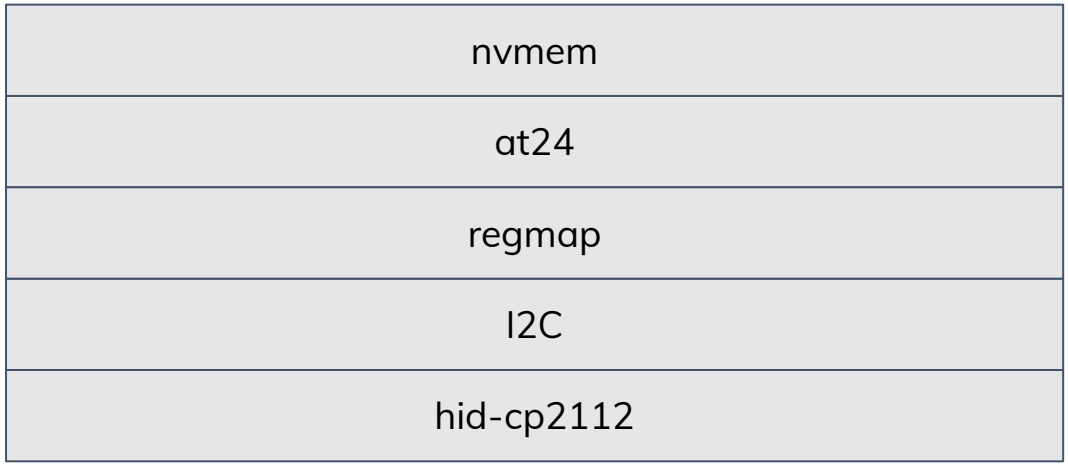
We go through so many abstraction layers...



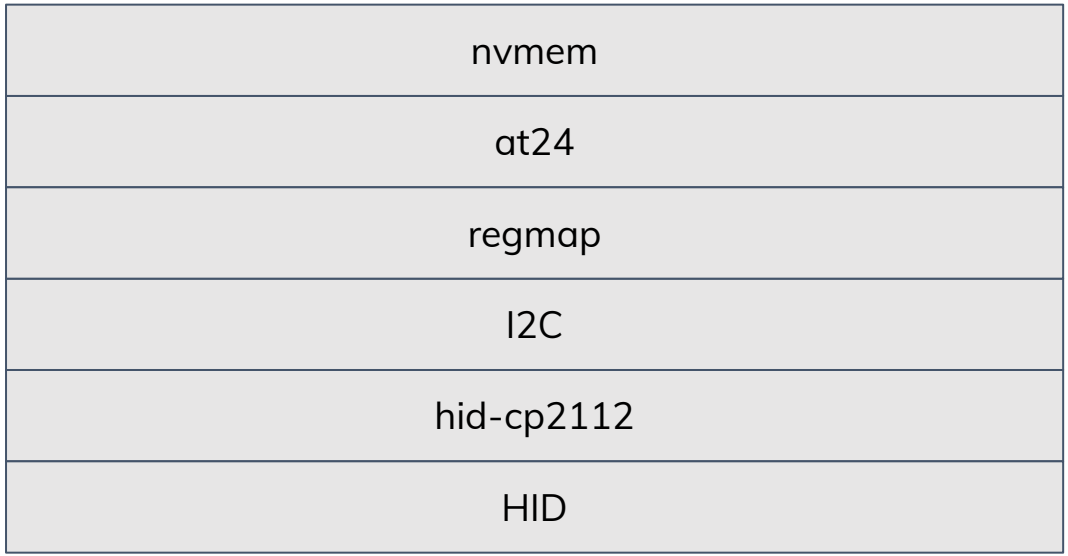
We go through so many abstraction layers...



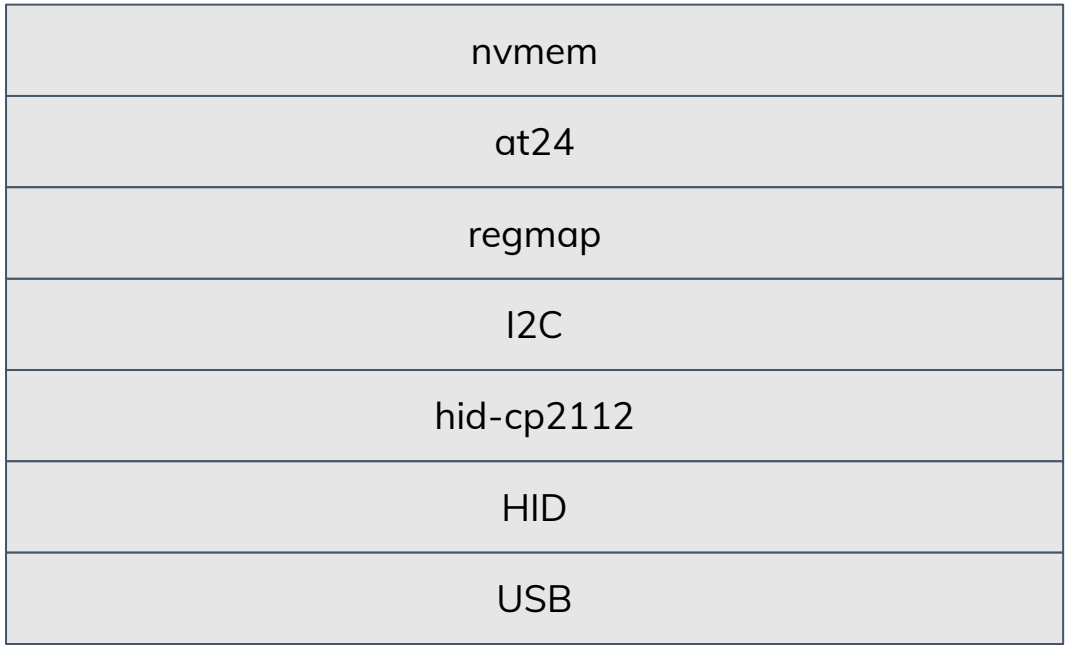
We go through so many abstraction layers...



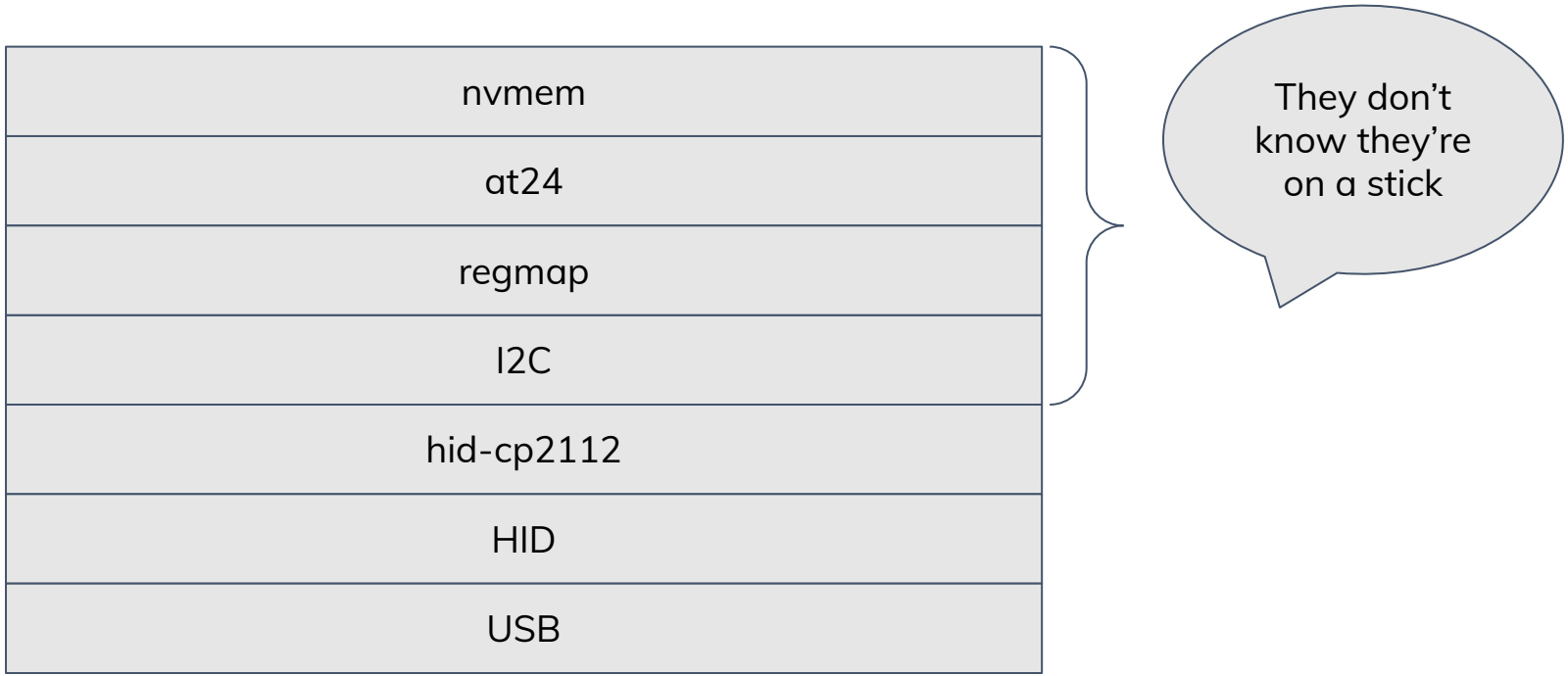
We go through so many abstraction layers...



We go through so many abstraction layers...

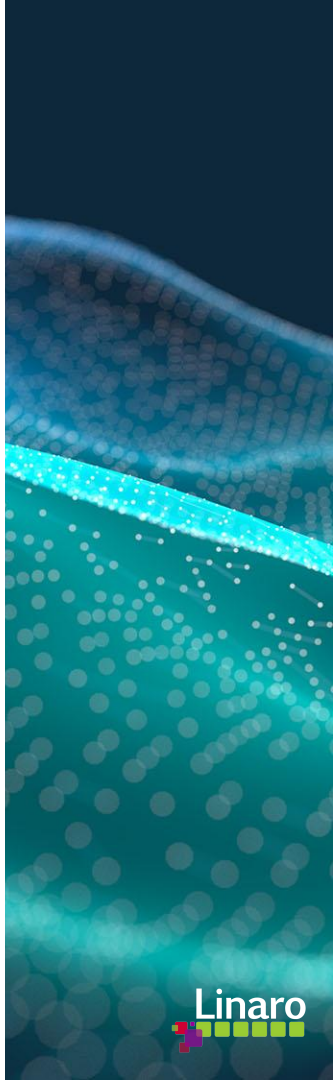


We go through so many abstraction layers...

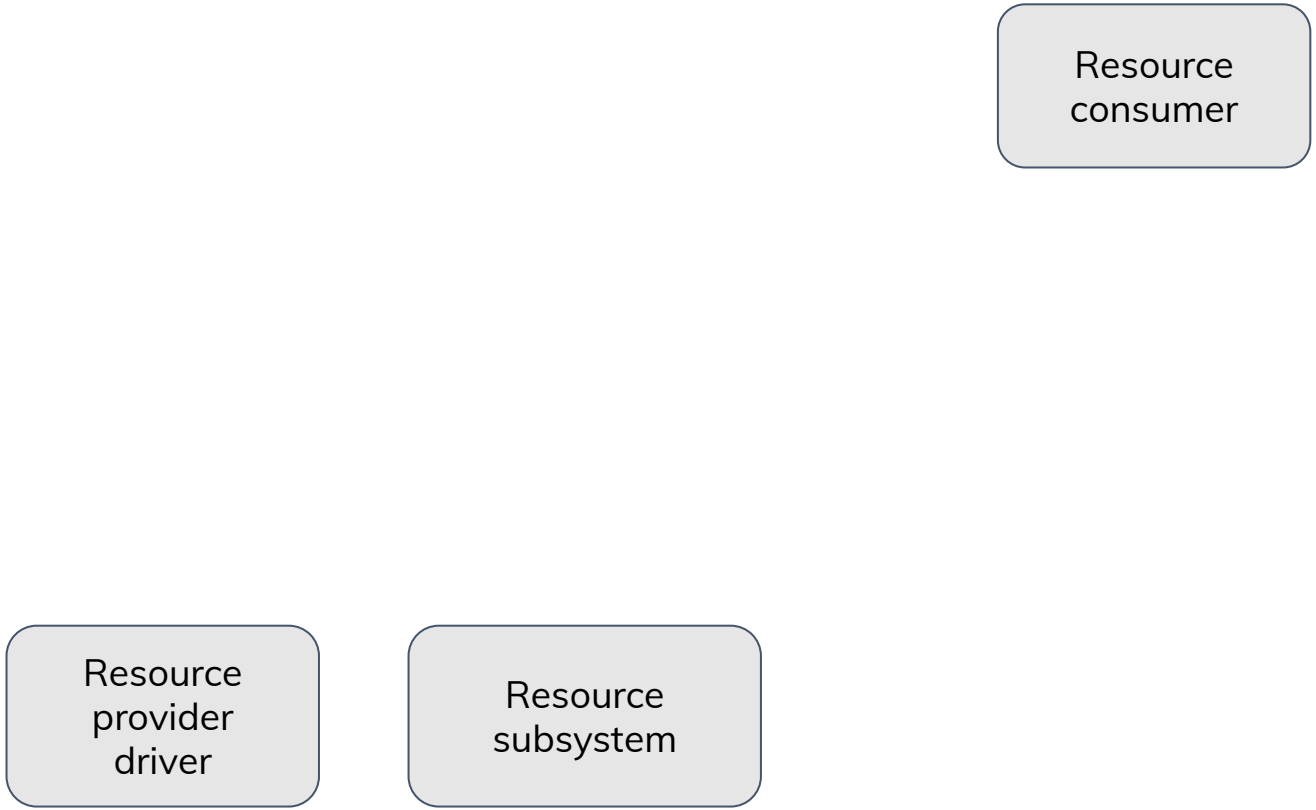


Let’s try a different approach

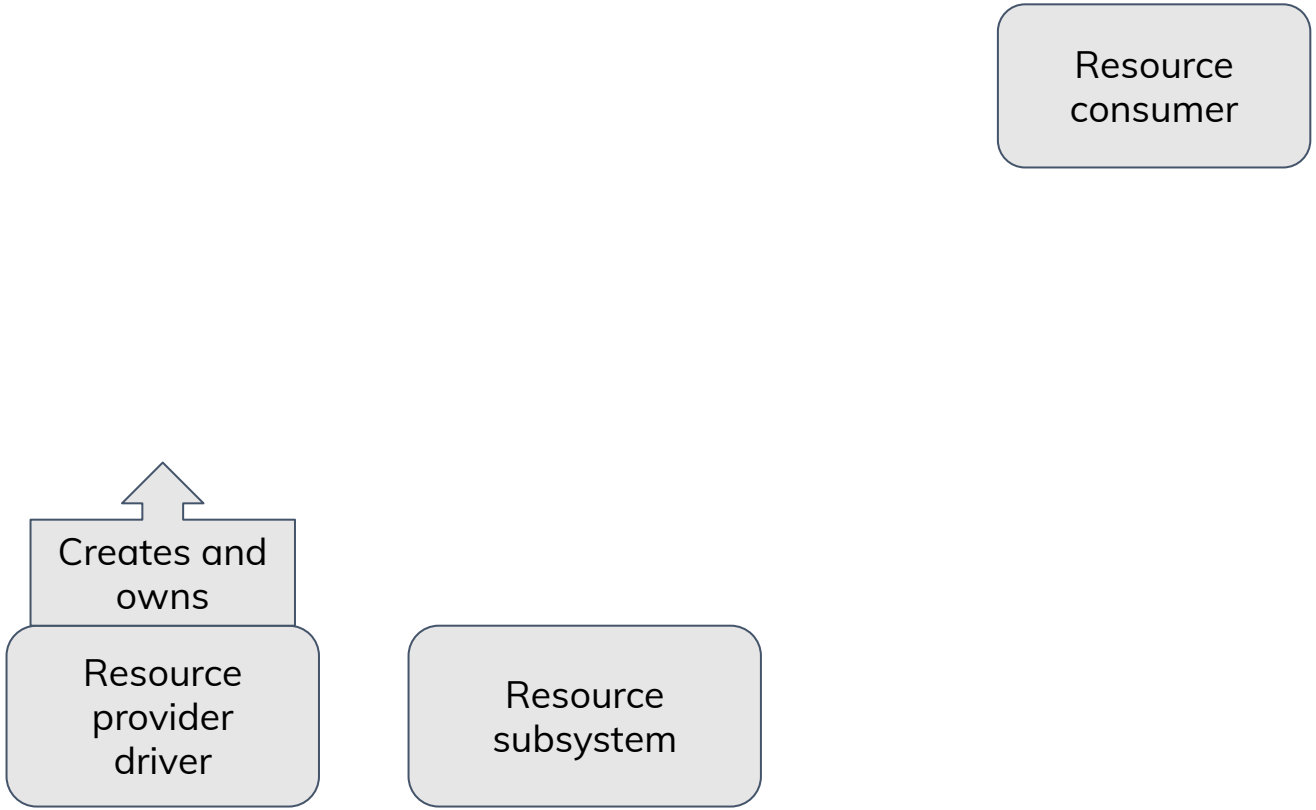
- Kernel complex resources are not like virtual memory. They can go away at any moment without the user knowing.
- Even if the subsystem doesn’t know it
- Caller of `kmalloc()` “owns” the allocated chunk
- Caller of `foo_get()` “references” the “foo” resource
- User should get a “weak” reference to a resource.
- User must release that reference but the underlying resource can already be gone.
- When it is gone, the API should gracefully communicate that to the caller



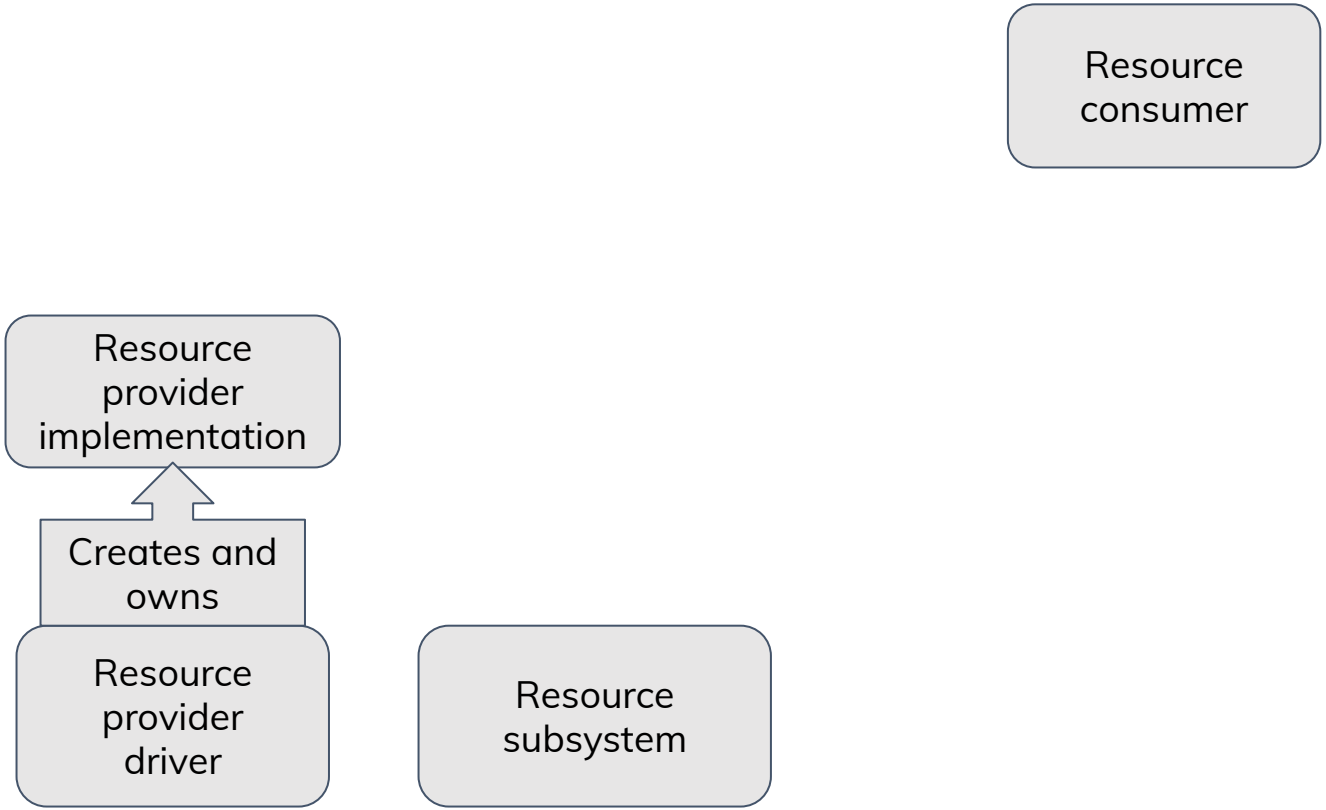
Common data model for resource providers



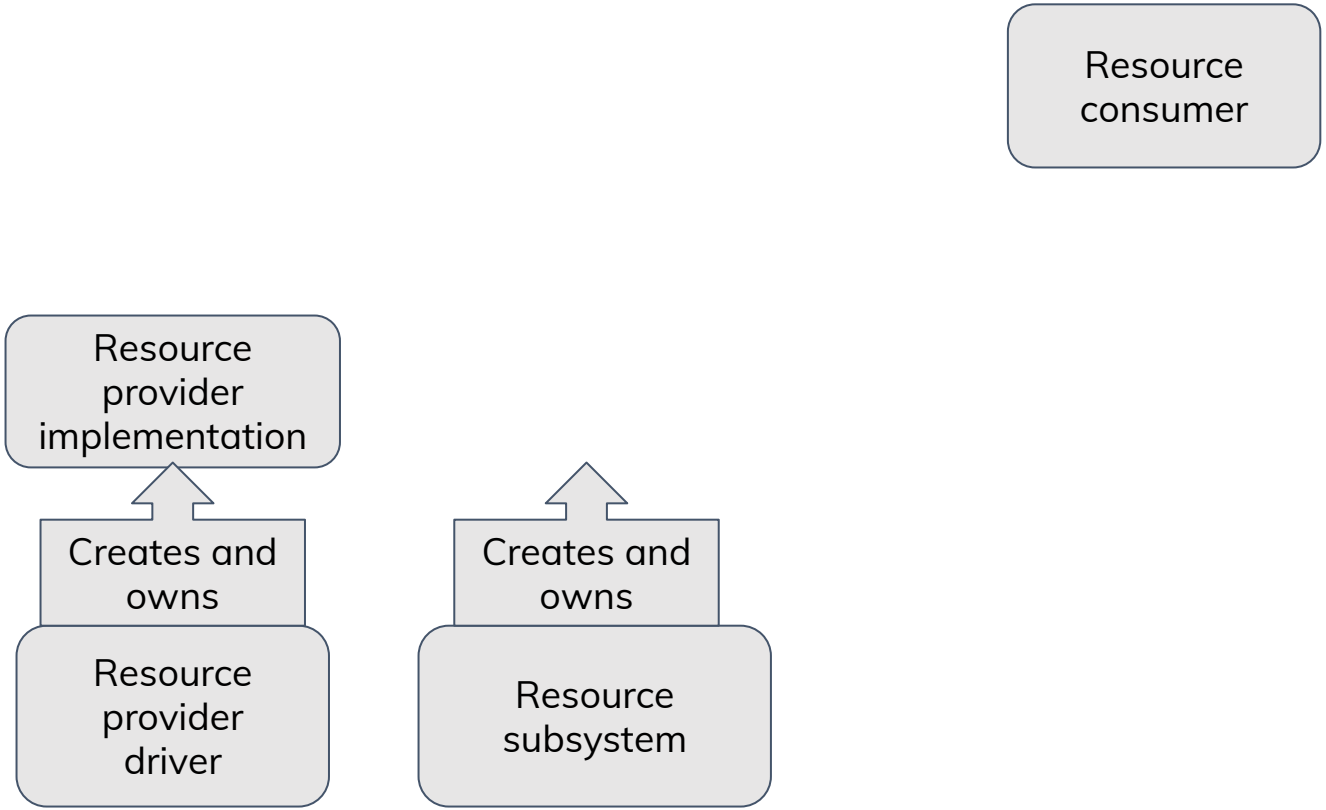
Common data model for resource providers



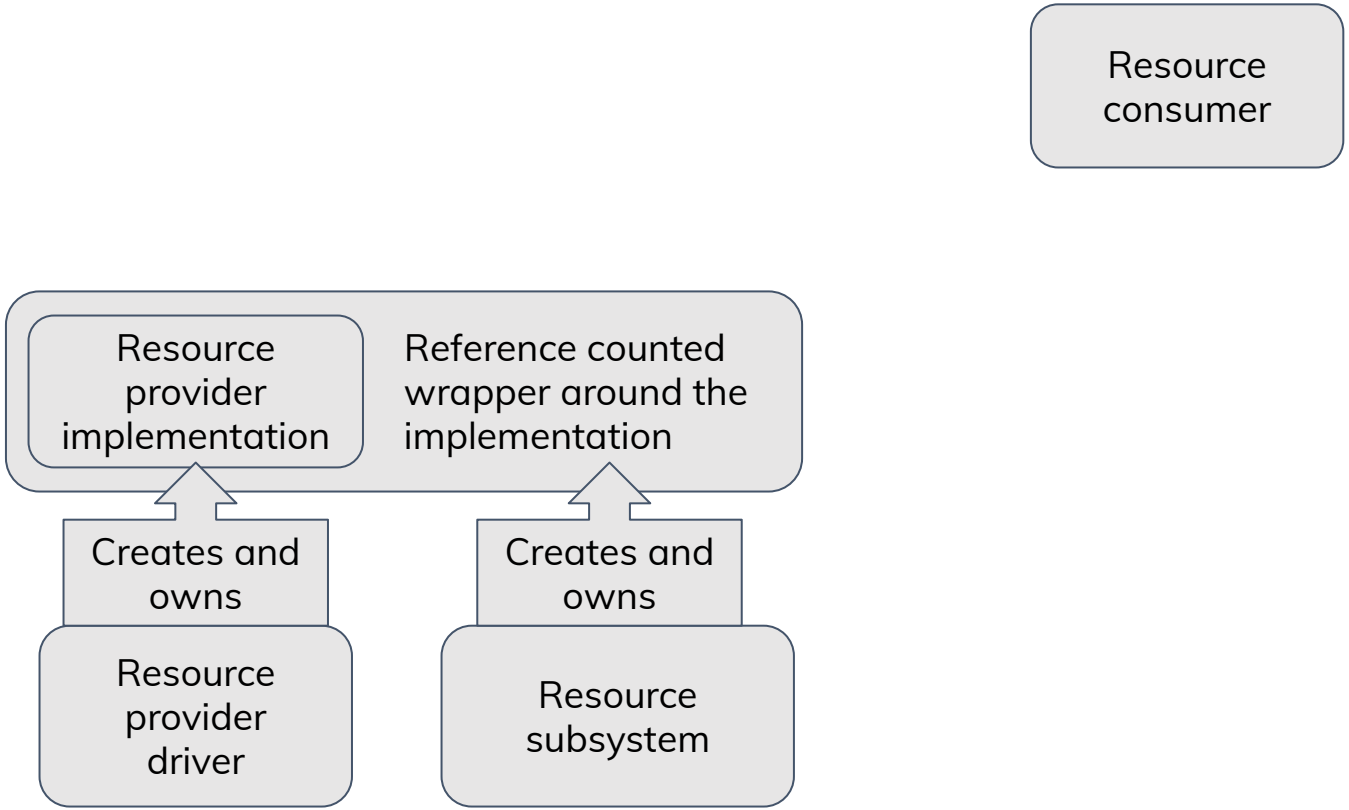
Common data model for resource providers



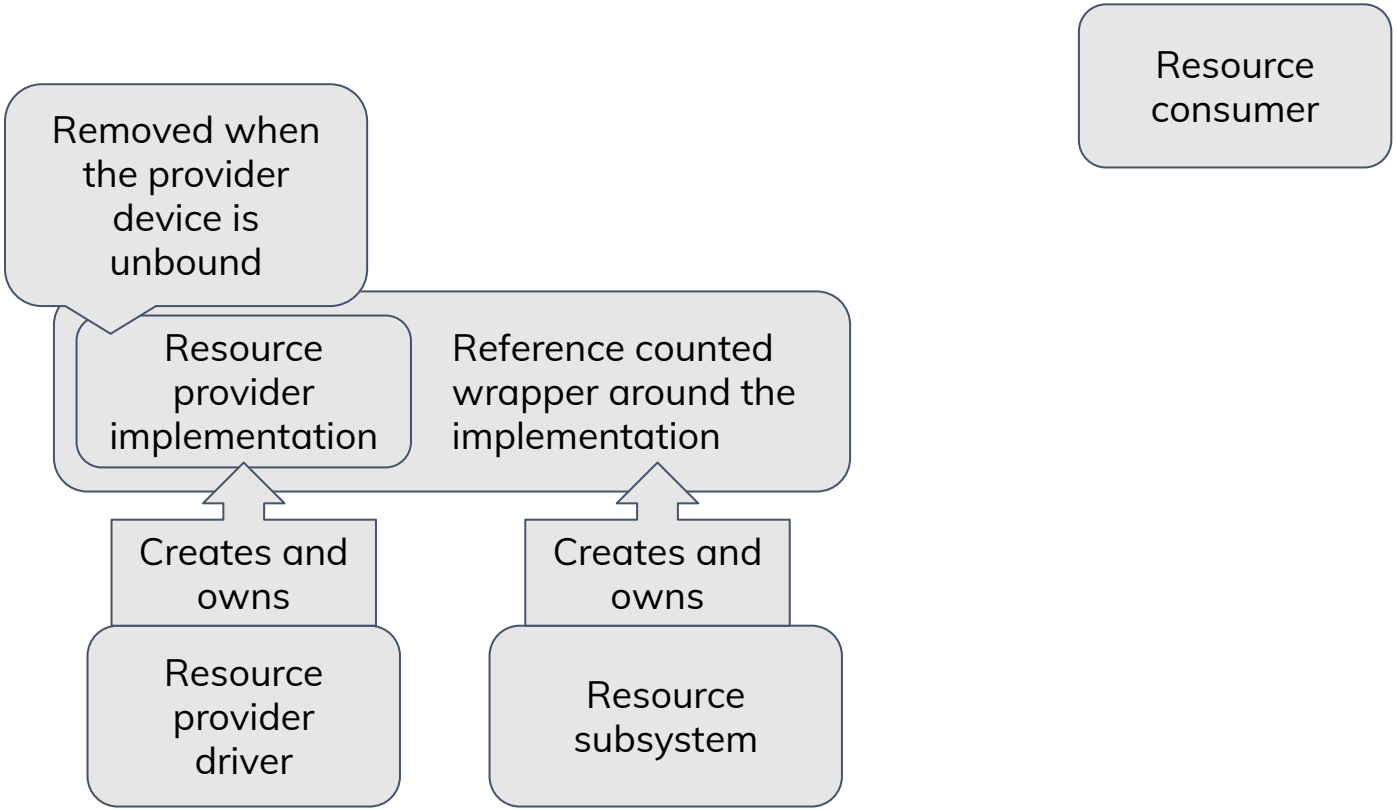
Common data model for resource providers



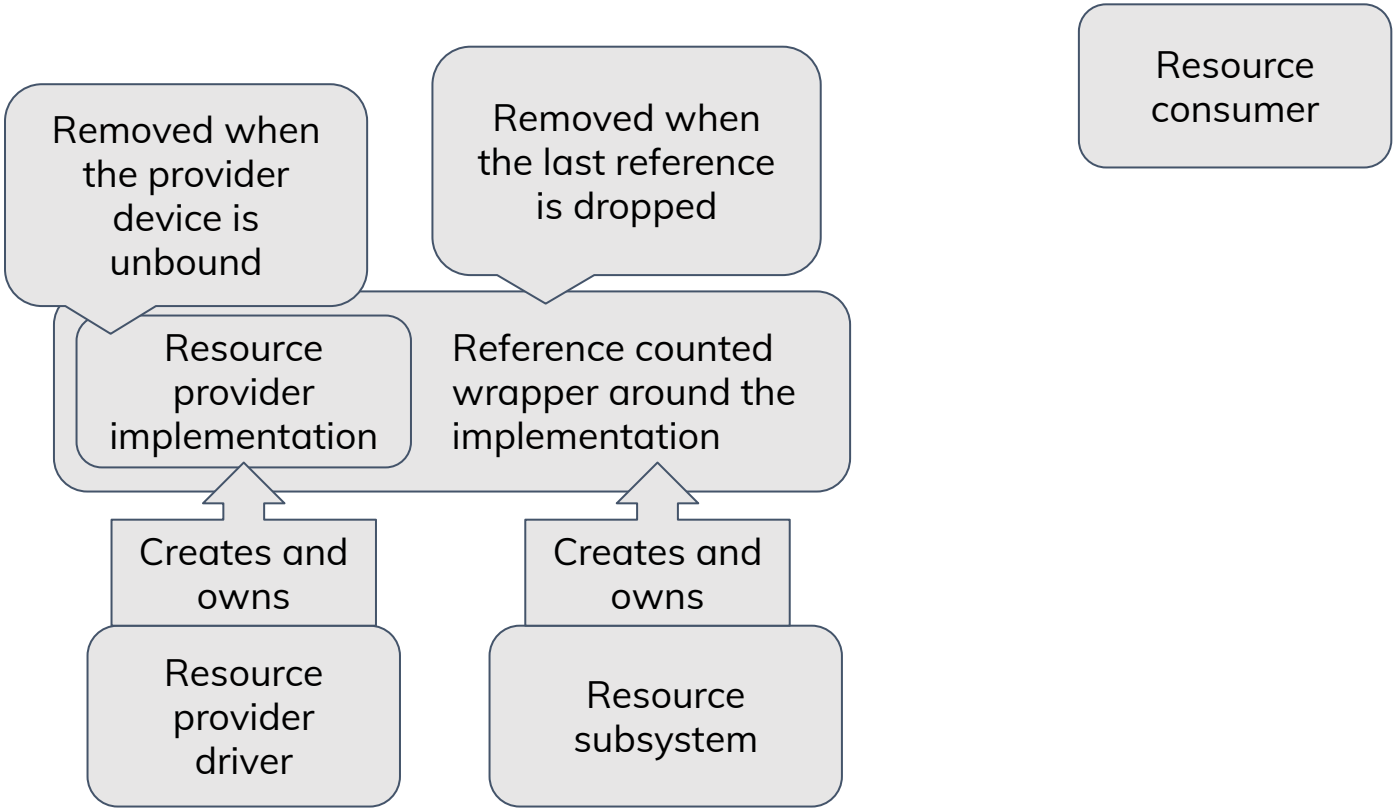
Common data model for resource providers



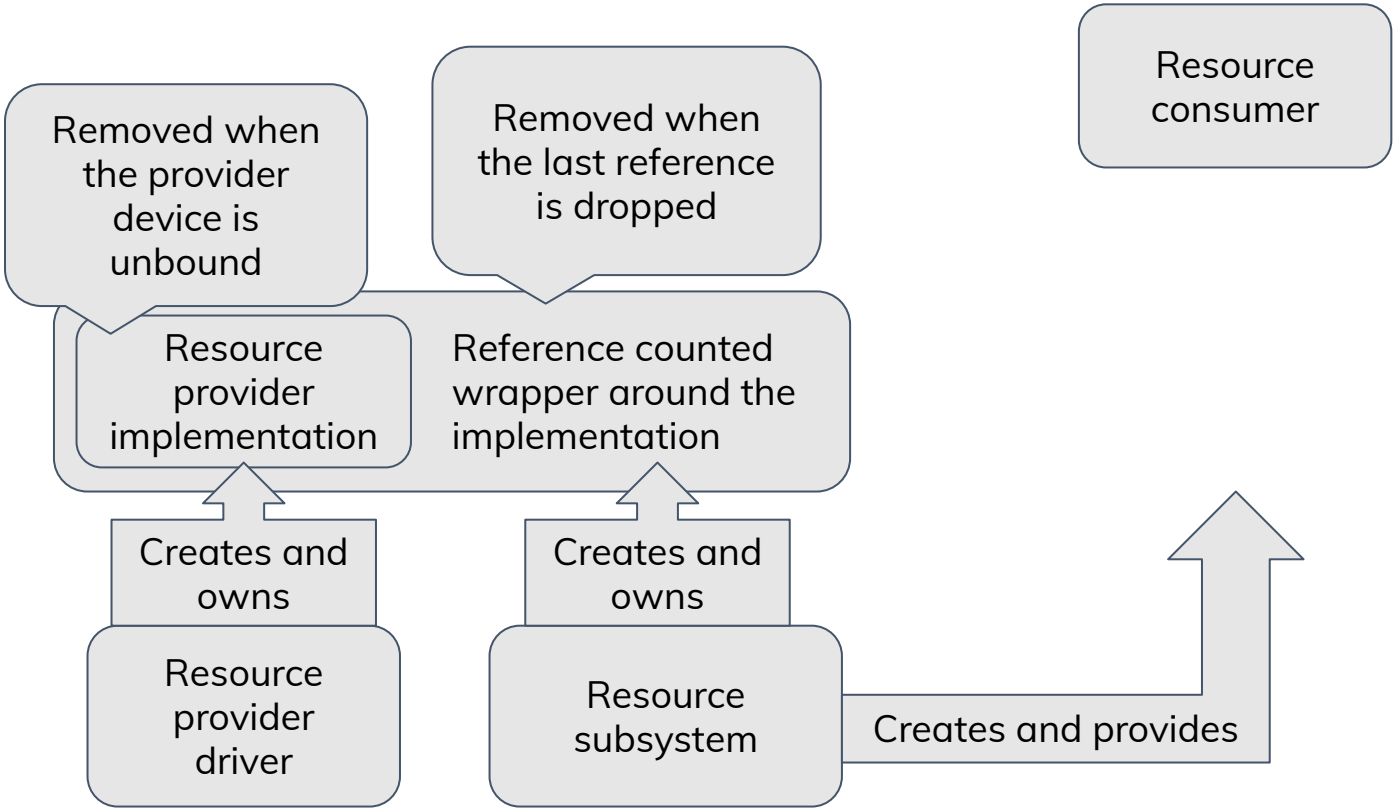
Common data model for resource providers



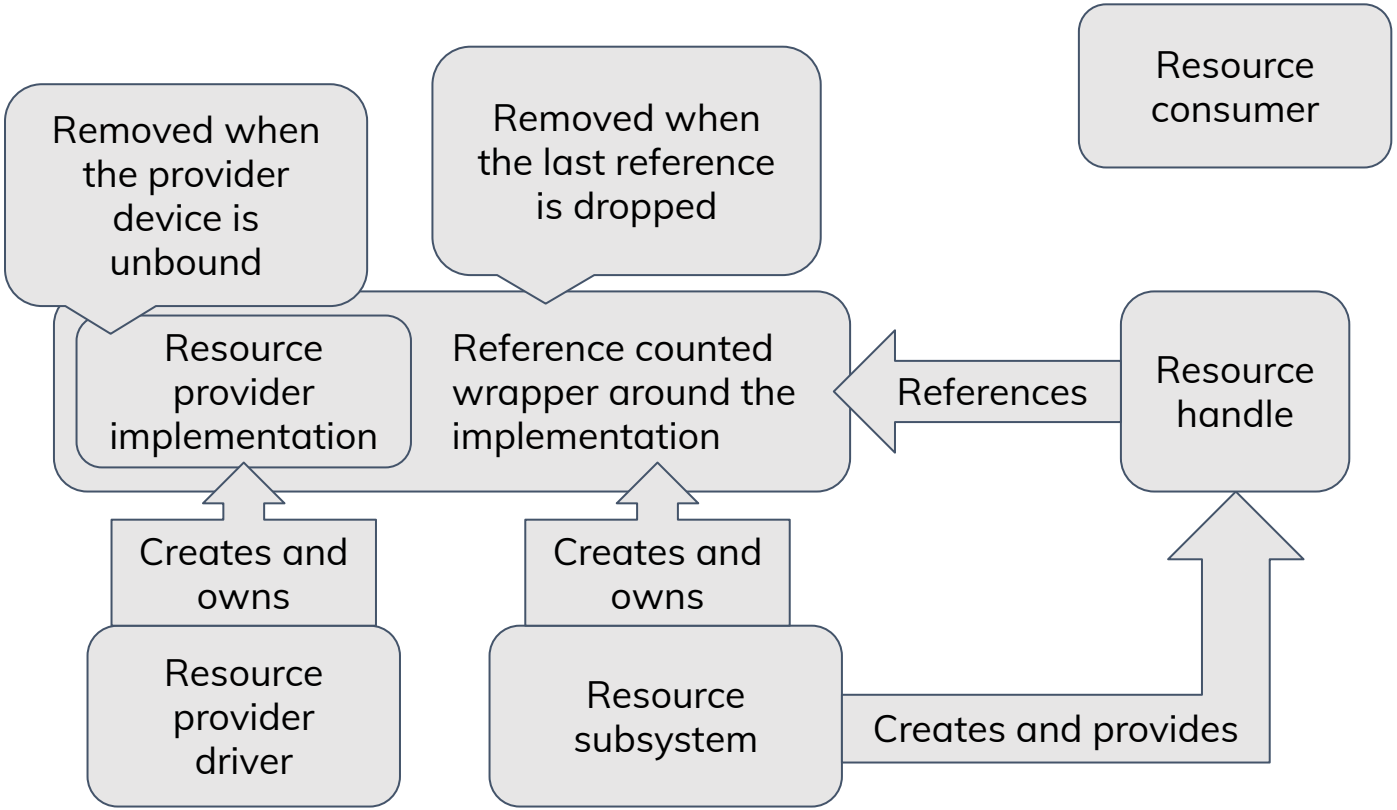
Common data model for resource providers



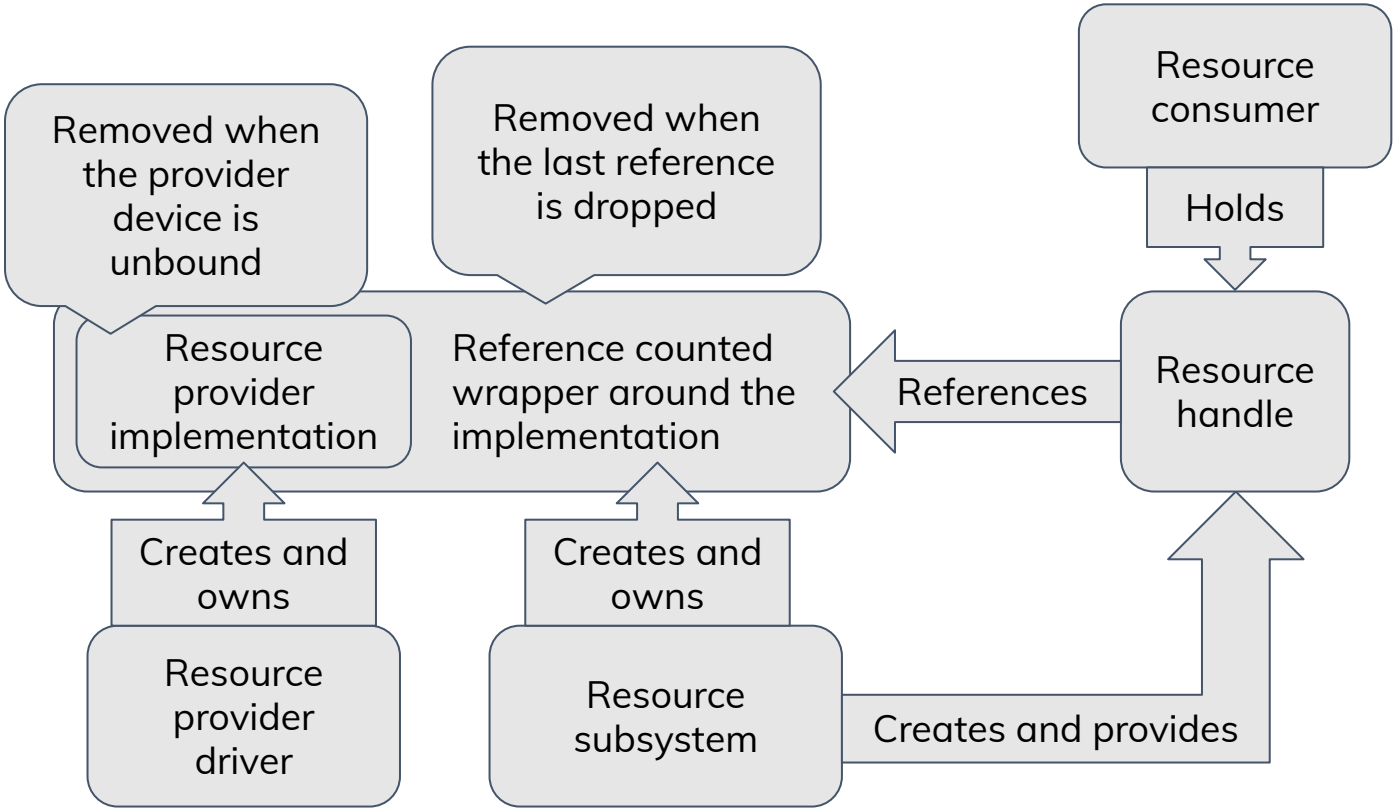
Common data model for resource providers



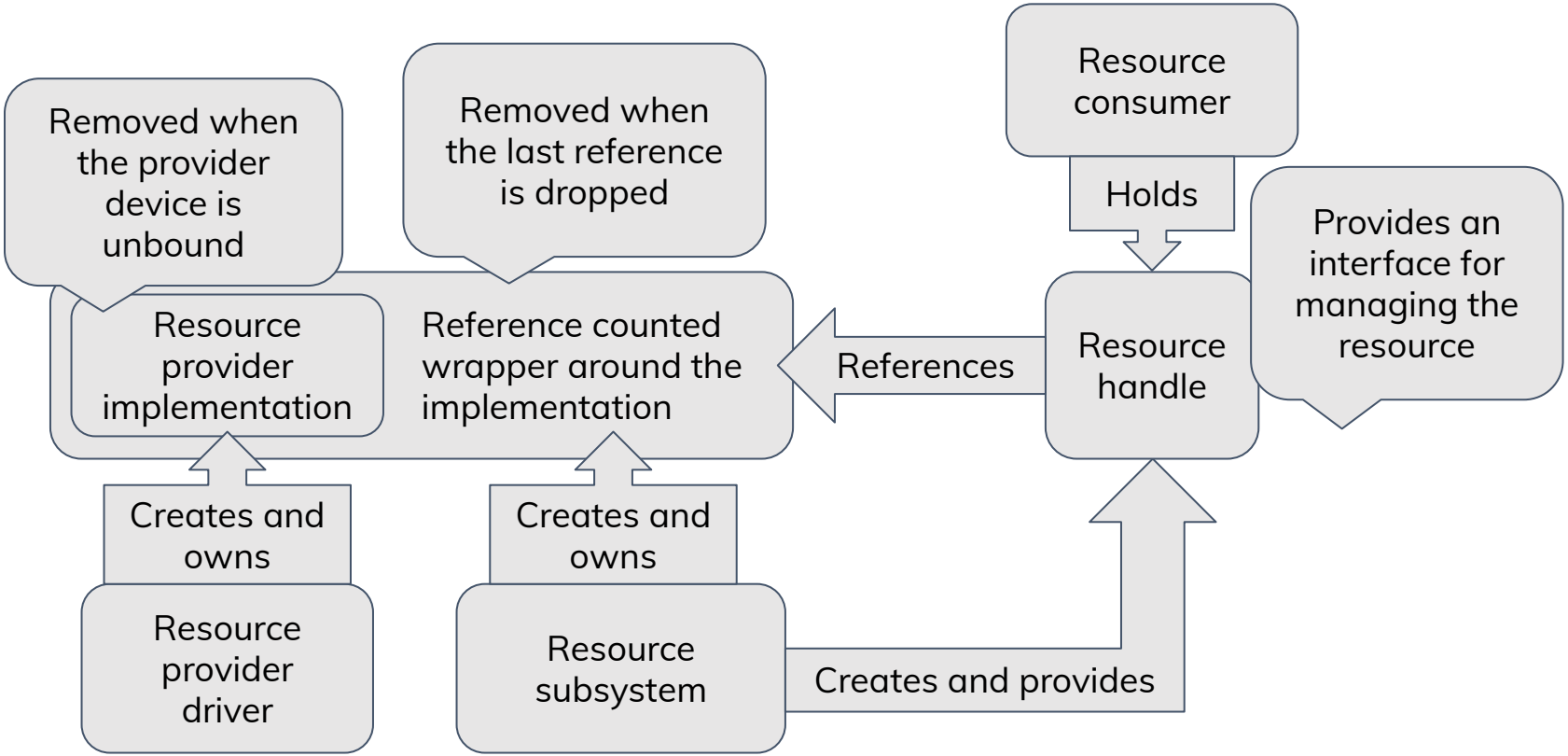
Common data model for resource providers



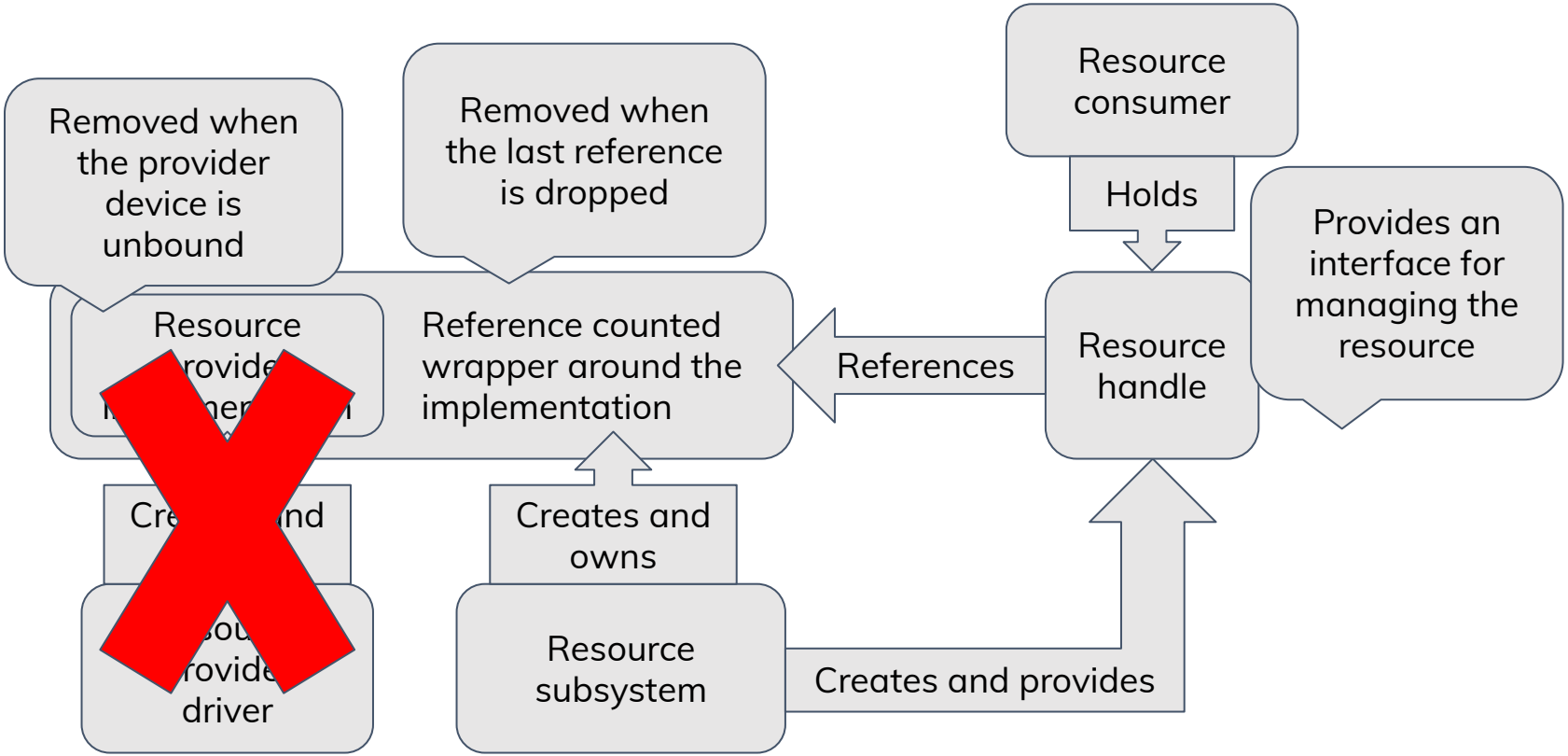
Common data model for resource providers



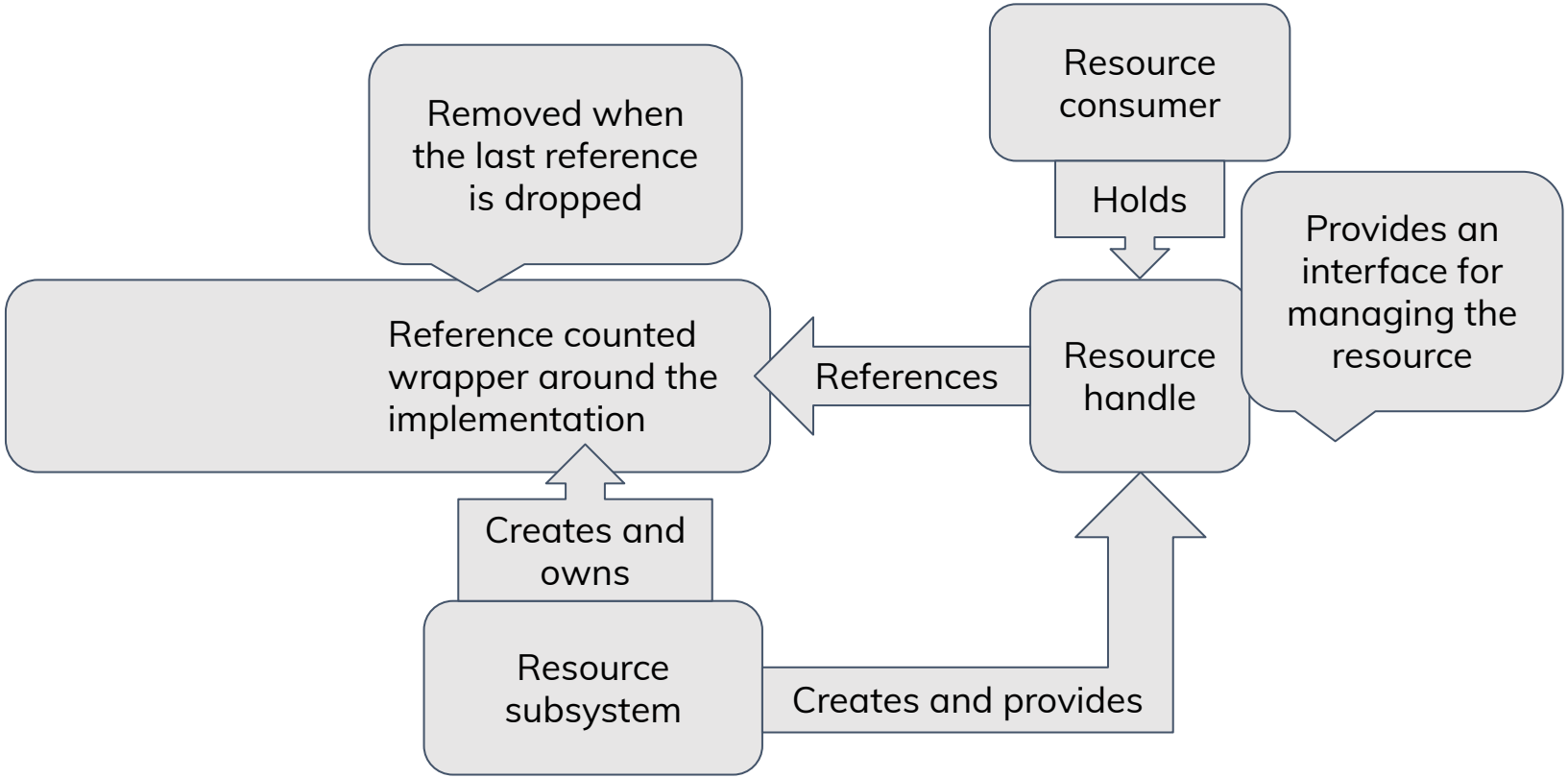
Common data model for resource providers



Common data model for resource providers



Common data model for resource providers



What about synchronization?

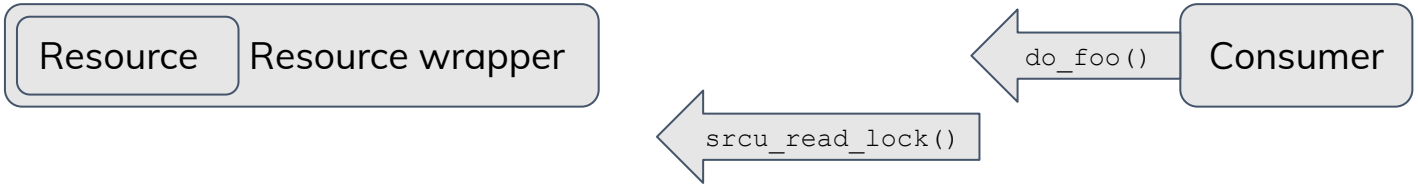
Protect the pointer to the implementation with SRCU

What about synchronization?

Protect the pointer to the implementation with SRCU

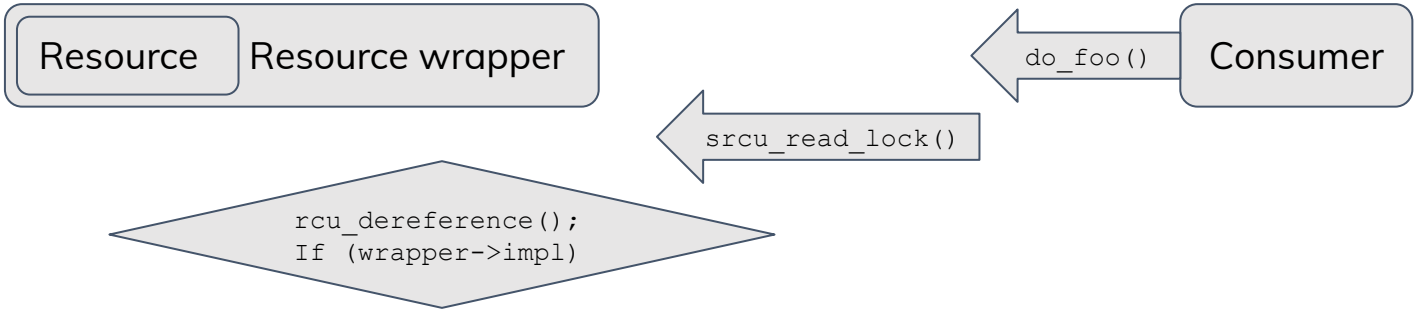
What about synchronization?

Protect the pointer to the implementation with SRCU



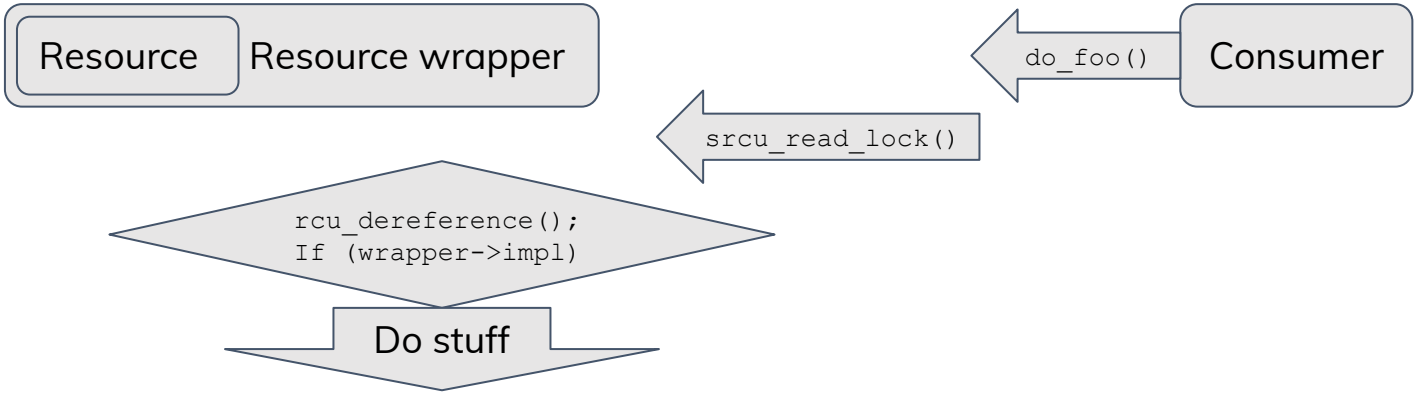
What about synchronization?

Protect the pointer to the implementation with SRCU



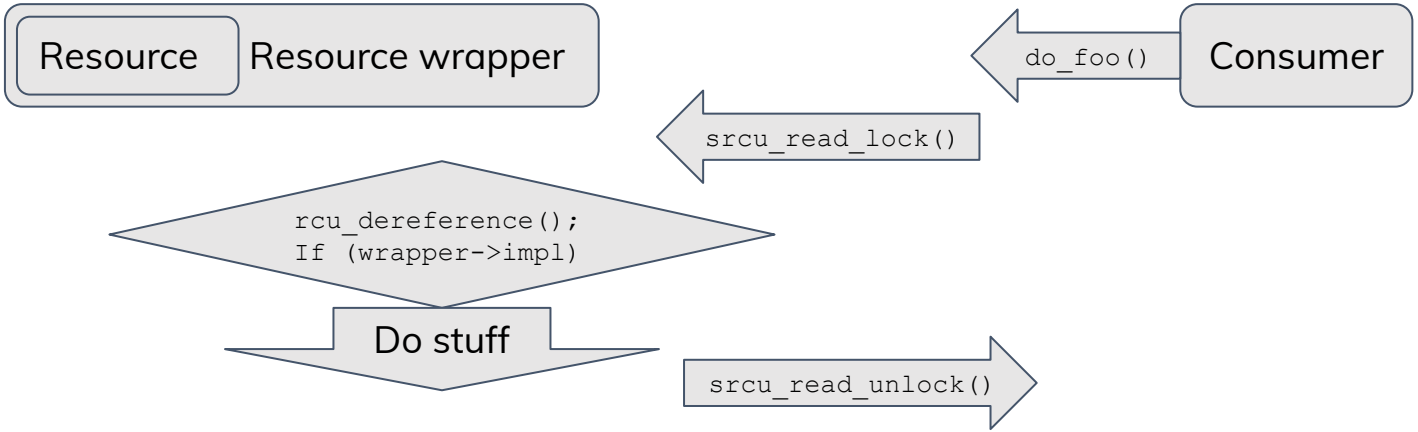
What about synchronization?

Protect the pointer to the implementation with SRCU



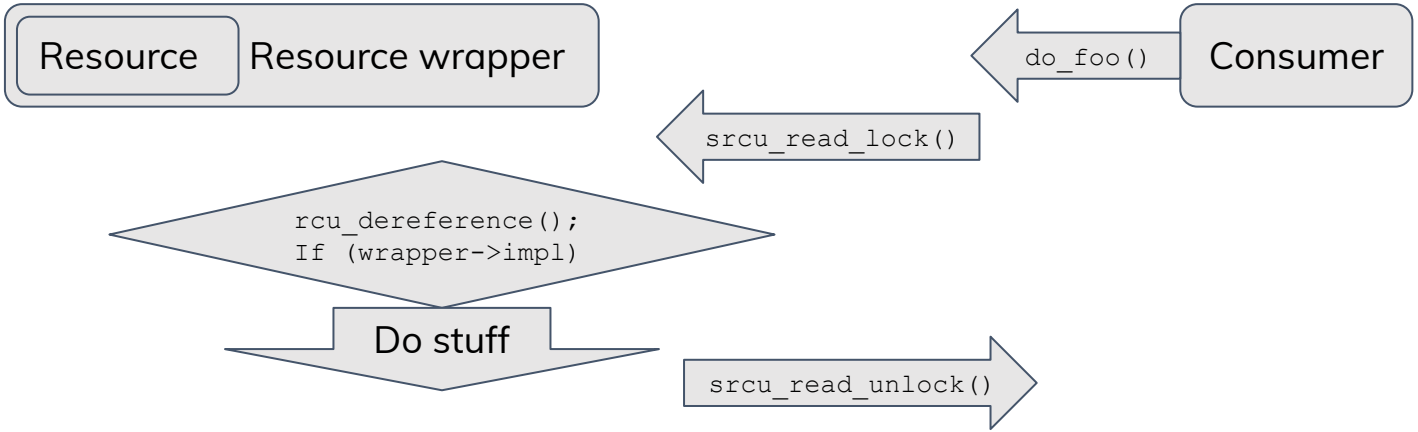
What about synchronization?

Protect the pointer to the implementation with SRCU



What about synchronization?

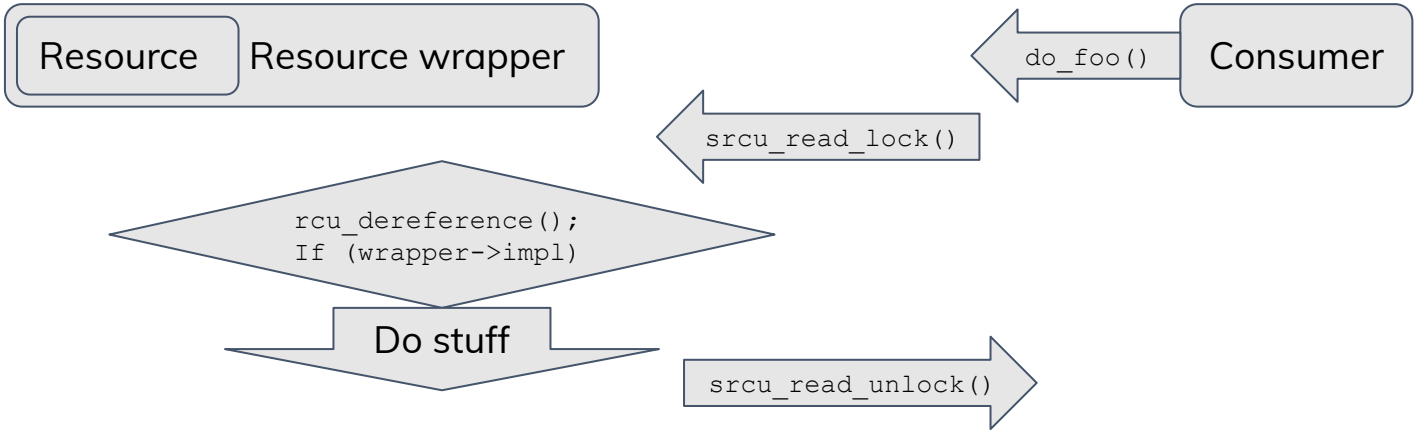
Protect the pointer to the implementation with SRCU



When provider is going down:

What about synchronization?

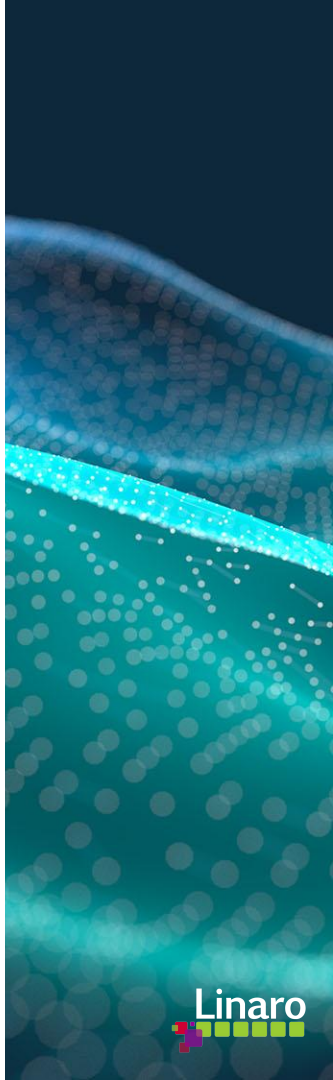
Protect the pointer to the implementation with SRCU



When provider is going down:

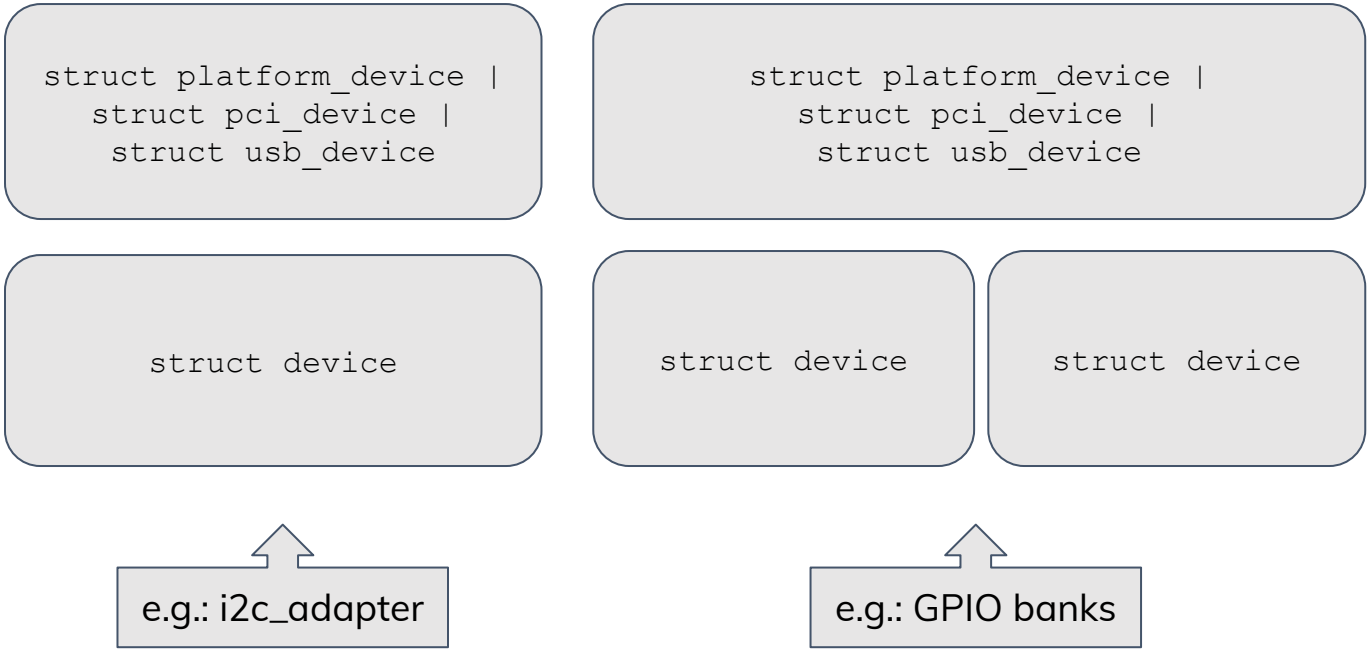
```
rcu_assign_pointer(handle->res, NULL);
synchronize_srcu()
```

Interlude



On `struct device`

- Common pattern in device drivers - physical and logical devices:



On `struct device`

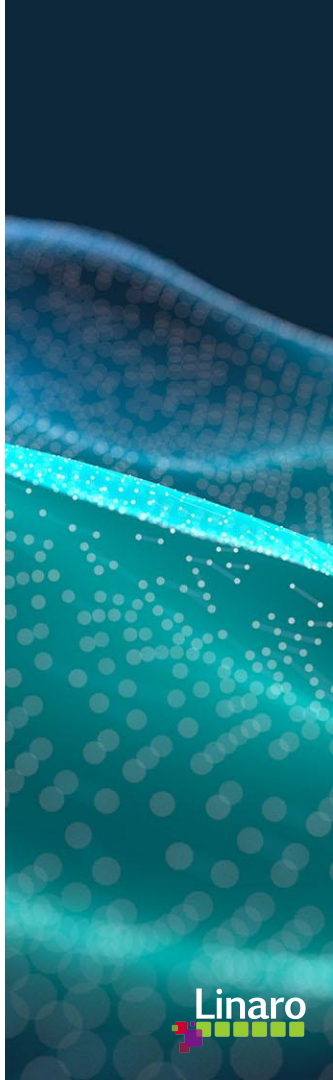
- Also a common pattern in device drivers:

```
struct foo {
    struct device dev;
    …
};
```

```
struct priv {
    struct foo foo;
    …
};
```

Driver

Subsystem



On `struct device`

- Also a common pattern in device drivers:

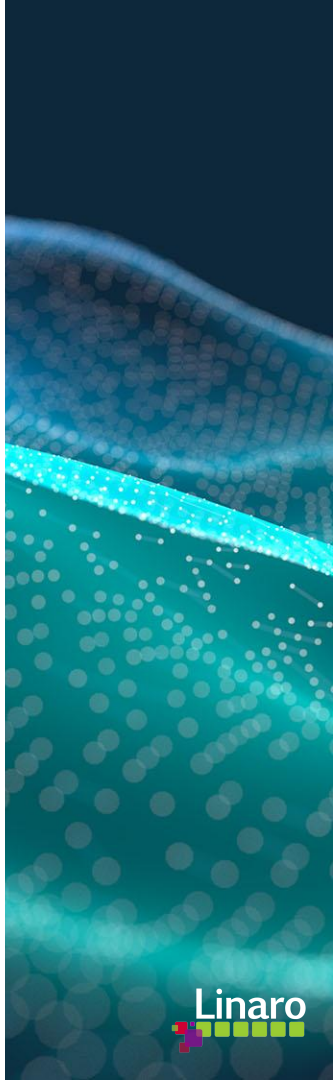
```
struct foo {
    struct device dev;
    …
};
```

```
struct priv {
    struct foo foo;
    …
};
```

Driver

```
priv = devm_kzalloc()
```

Subsystem

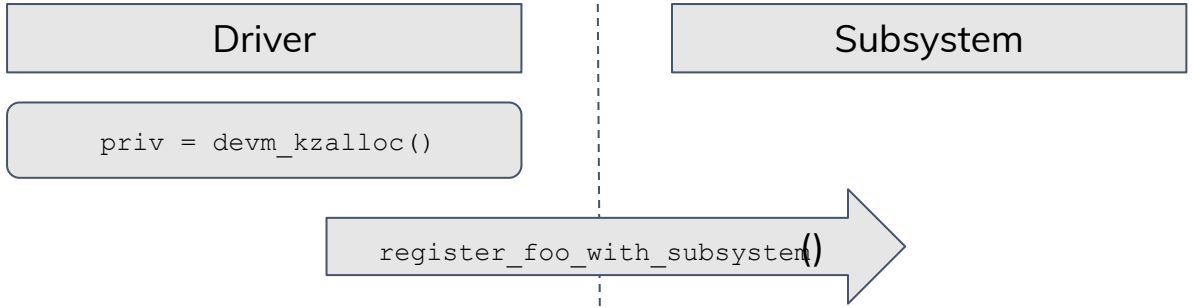


On `struct device`

- Also a common pattern in device drivers:

```
struct foo {
    struct device dev;
    …
};
```

```
struct priv {
    struct foo foo;
    …
};
```

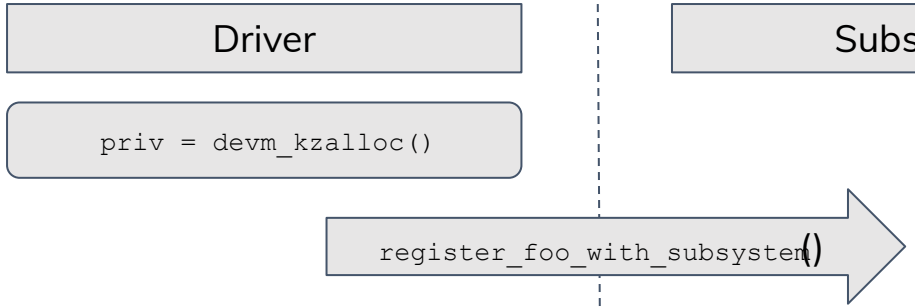


On `struct device`

- Also a common pattern in device drivers:

```
struct foo {
    struct device dev;
    …
};
```

```
struct priv {
    struct foo foo;
    …
};
```

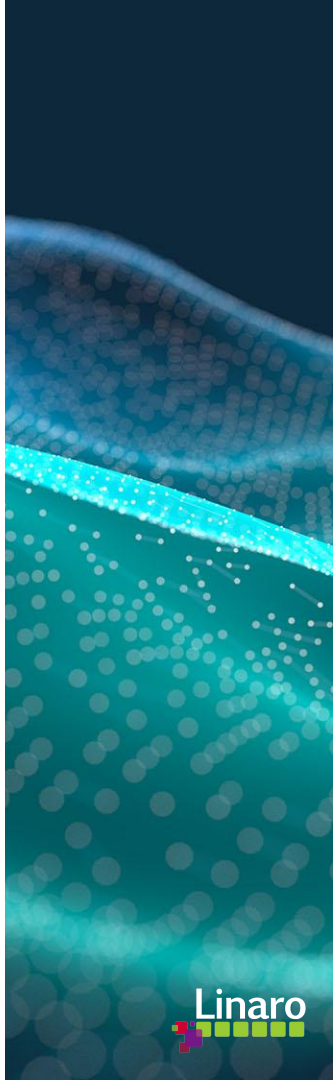


- foo will be freed when the device is unbound but consumers may still hold references to `struct device`!

On `struct device`

- Subsystems will bravely venture out to fix problems they create by this approach
- Some hand over foo entirely to the subsystem who'll be in charge of its life-time
  - alloc_foo() + register_foo() approach
  - Breaks life-time logic - the driver allocates the object but is not responsible for freeing it
- Others do crazy things like:
  - block on completion for the last reference to be released when the provider device is unbound (I'm looking at you i[23]c)
  - have a field in subsystem-specific structures e.g. `bool managed` that's set to true if the structure was allocated with devres which defeats the purpose of devres (!)
  - expect the drivers to populate the device release callback (!?)

- Of course many subsystems will just act like the problem doesn't exist and produce fireworks on unbind



Wolfram's preferred solution:

Driver

```
struct foo {
    struct device dev;
    …
};
```

```
struct priv {
    struct foo *foo;
    …
};
```

Subsystem

Wolfram's preferred solution:

Driver

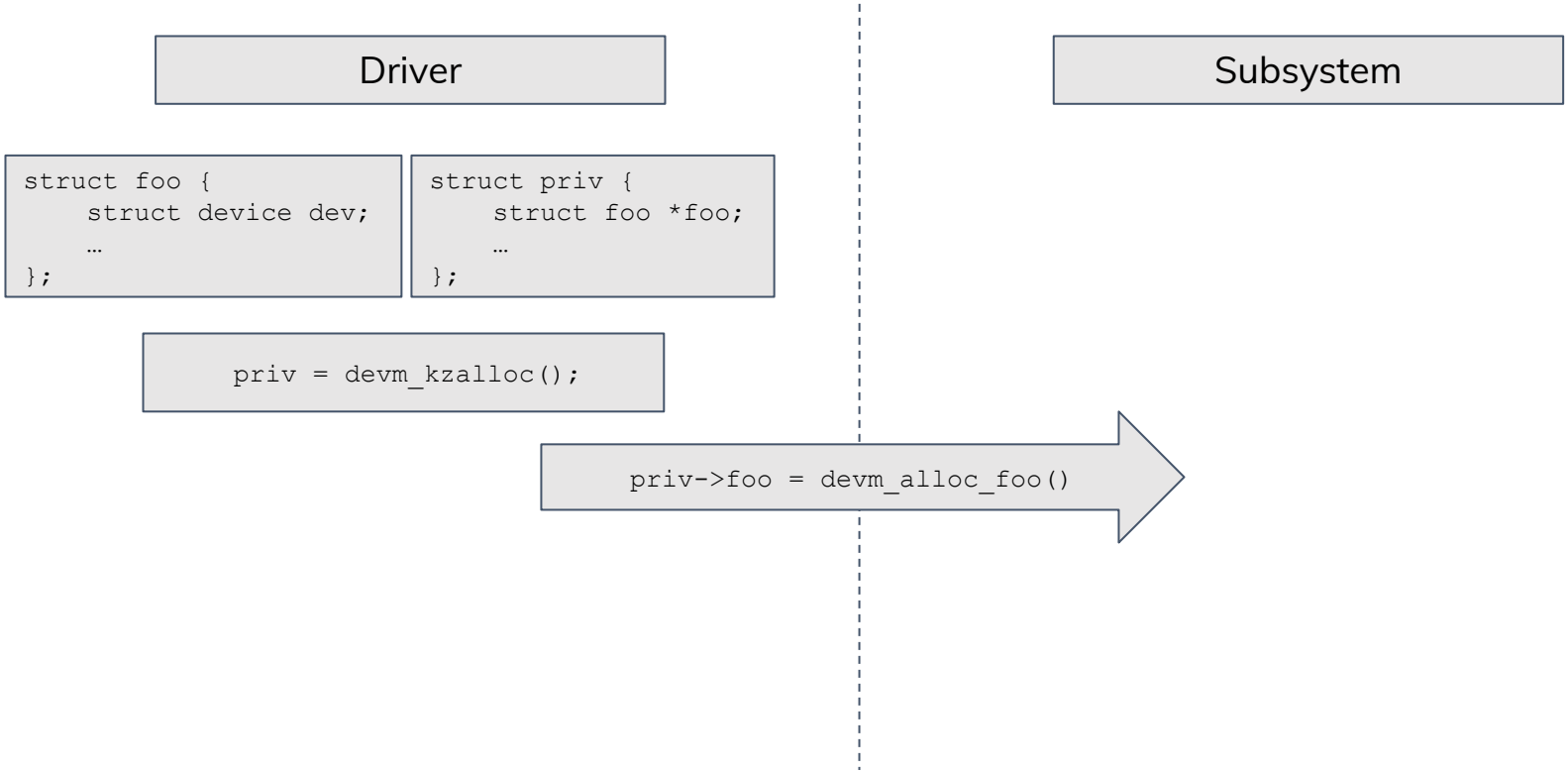
```
struct foo {
    struct device dev;
    …
};
```

```
struct priv {
    struct foo *foo;
    …
};
```

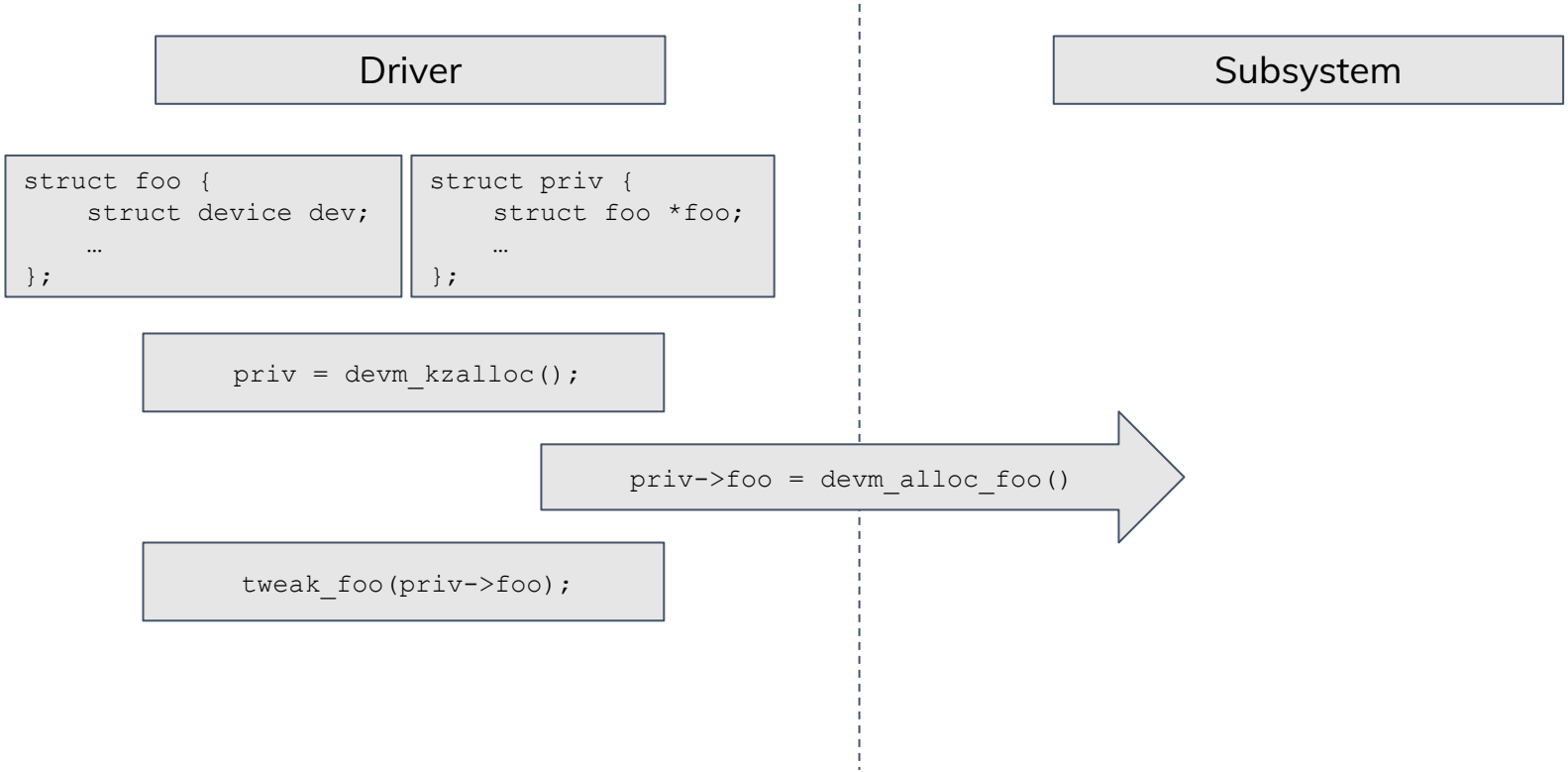
```
priv = devm_kzalloc();
```

Subsystem

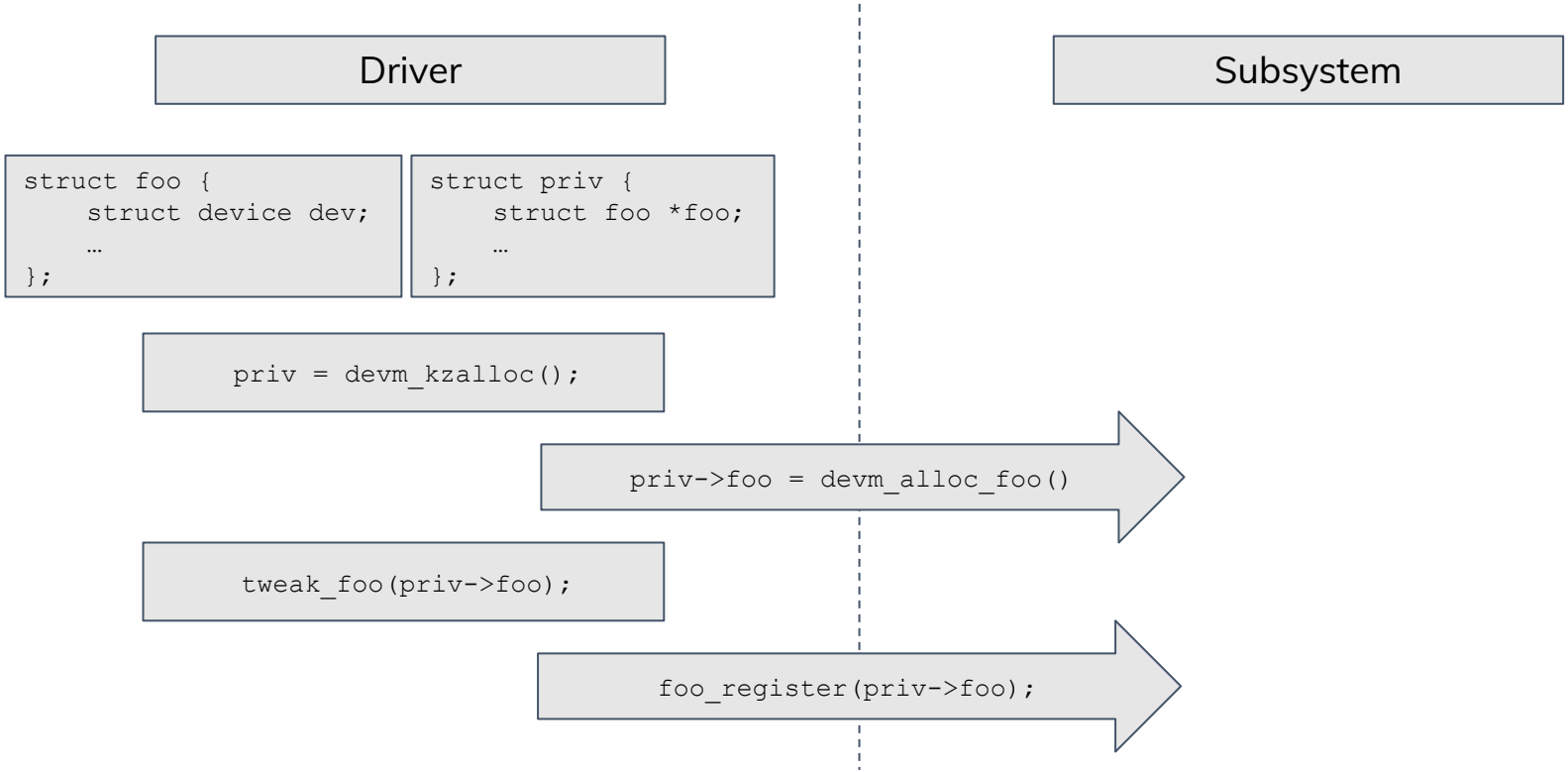
Wolfram's preferred solution:



Wolfram's preferred solution:



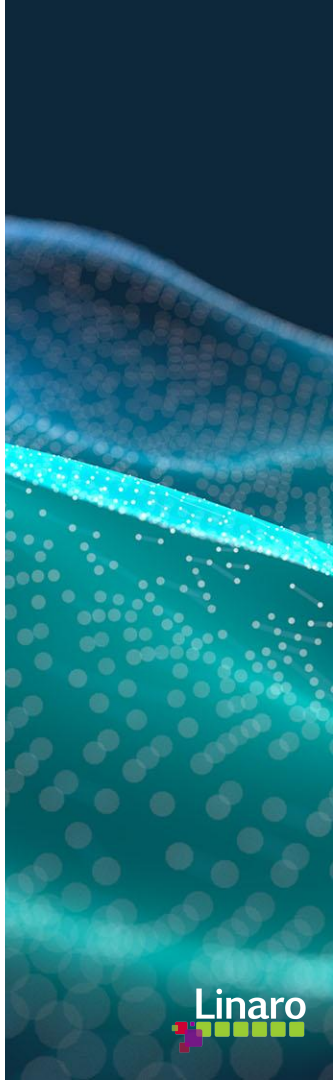
Wolfram's preferred solution:



Linaro

Wolfram's preferred solution:

- Asymmetric logic: drivers should be in charge of releasing resources they acquire, IOW every resource acquisition should be accompanied by the corresponding releasing within the same scope: `alloc() -> free(), get() -> put(), ref() -> unref()`
- Unnecessarily convoluted: drivers almost never need to access the logical `struct device`



Let’s grill two chickens on the same skewer or something

```
struct foo {
    struct foo_ops ops;
    …
};
```

```
struct priv {
    struct foo foo;
    …
};
```

Driver

```
struct foo_wrapper {
    struct device dev;
    struct foo __rcu *impl;
    …
};
```

Subsystem

Let’s grill two chickens on the same skewer or something

```
struct foo {
    struct foo_ops ops;
    …
};
```

```
struct priv {
    struct foo foo;
    …
};
```

```
struct foo_wrapper {
    struct device dev;
    struct foo __rcu *impl;
    …
};
```

Driver

```
priv = devm_kzalloc()
```

Subsystem

Let’s grill two chickens on the same skewer or something

```
struct foo {
    struct foo_ops ops;
    …
};
```

```
struct priv {
    struct foo foo;
    …
};
```

```
struct foo_wrapper {
    struct device dev;
    struct foo __rcu *impl;
    …
};
```

Driver

```
priv = devm_kzalloc()
```

```
priv->foo.ops = { };
```

Subsystem

Let’s grill two chickens on the same skewer or something

```
struct foo {
    struct foo_ops ops;
    …
};
```

```
struct priv {
    struct foo foo;
    …
};
```

```
struct foo_wrapper {
    struct device dev;
    struct foo __rcu *impl;
    …
};
```

Driver

Subsystem

```
priv = devm_kzalloc()
```

```
priv->foo.ops = { };
```

```
register_foo_with_subsystem(&priv->foo)
```

Let’s grill two chickens on the same skewer or something

```
struct foo {
    struct foo_ops ops;
    …
};
```

```
struct priv {
    struct foo foo;
    …
};
```

```
struct foo_wrapper {
    struct device dev;
    struct foo __rcu *impl;
    …
};
```

Driver

Subsystem

```
priv = devm_kzalloc()
```

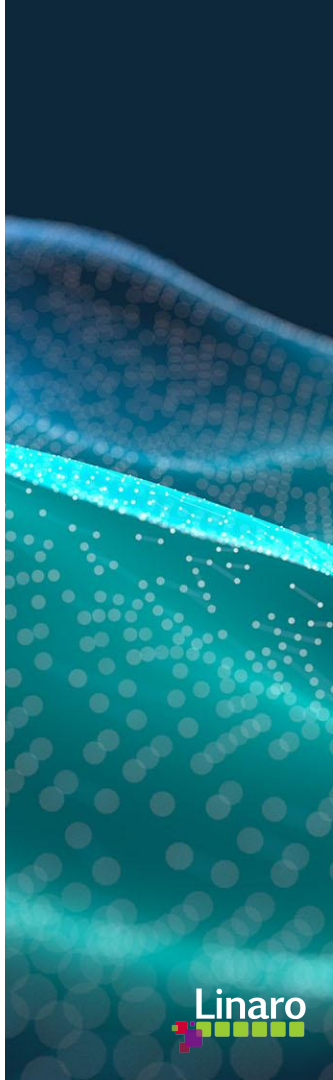
```
priv->foo.ops = { };
```

```
register_foo_with_subsystem(&priv->foo)
```

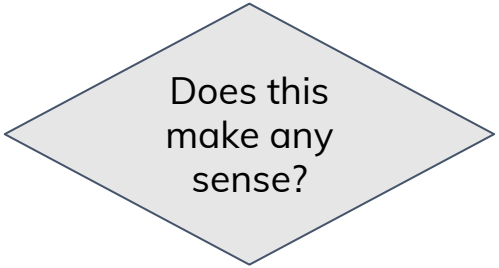
```
wrp = make_foo_wrapper(foo)
```

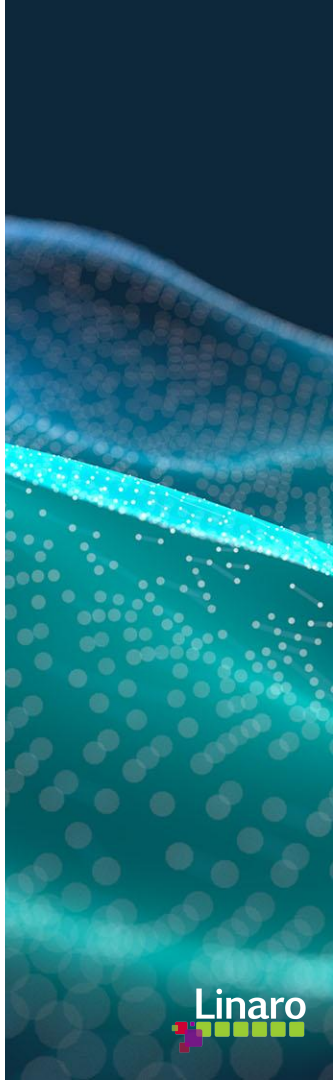
Linaro

Interlude

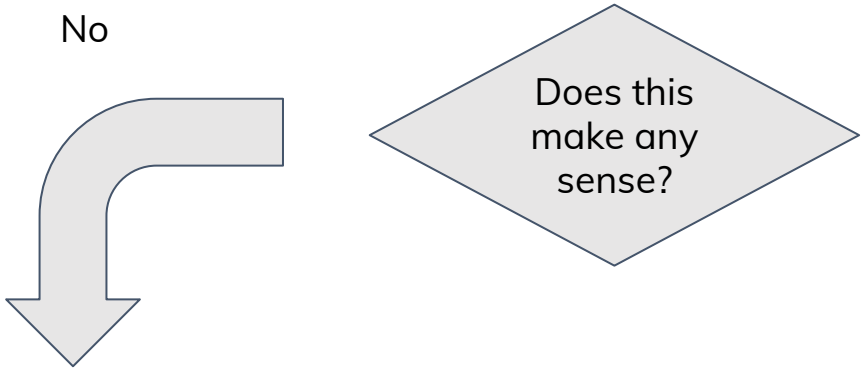


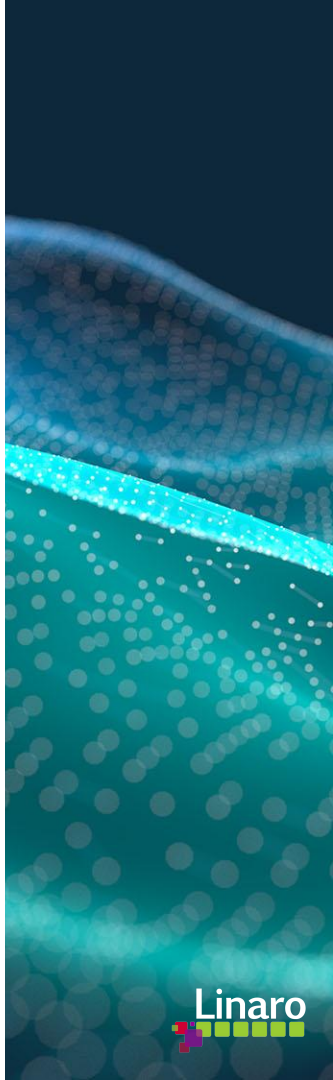
The way forward



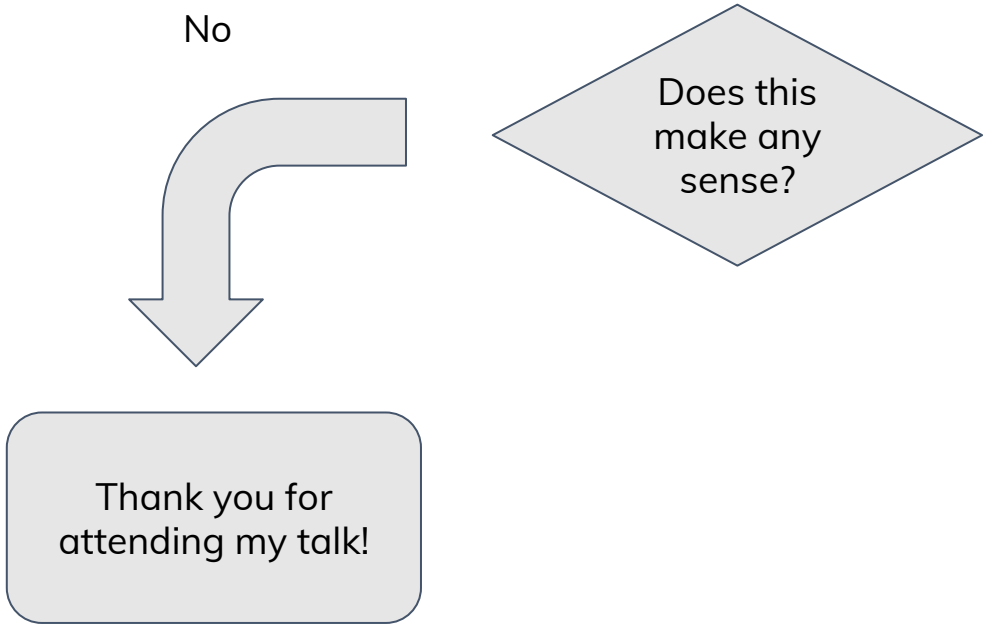


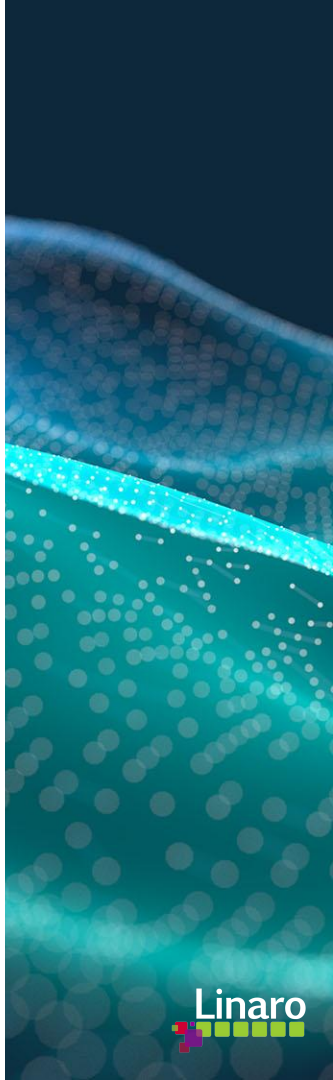
The way forward



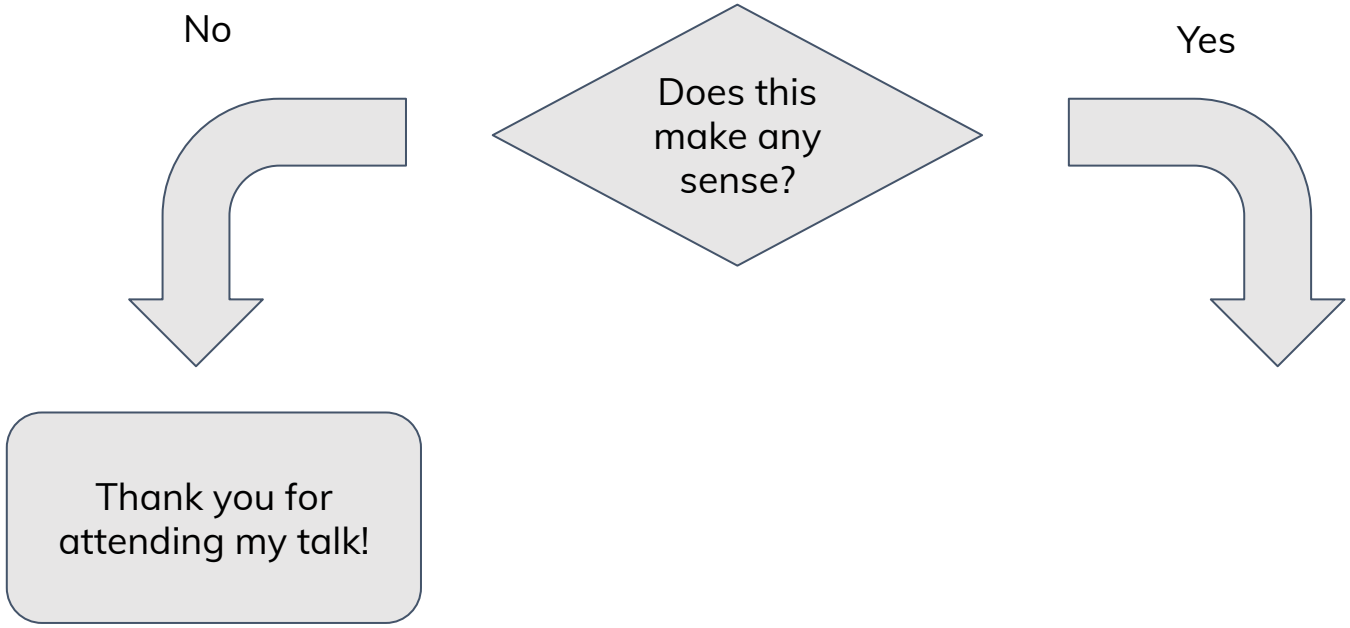


The way forward

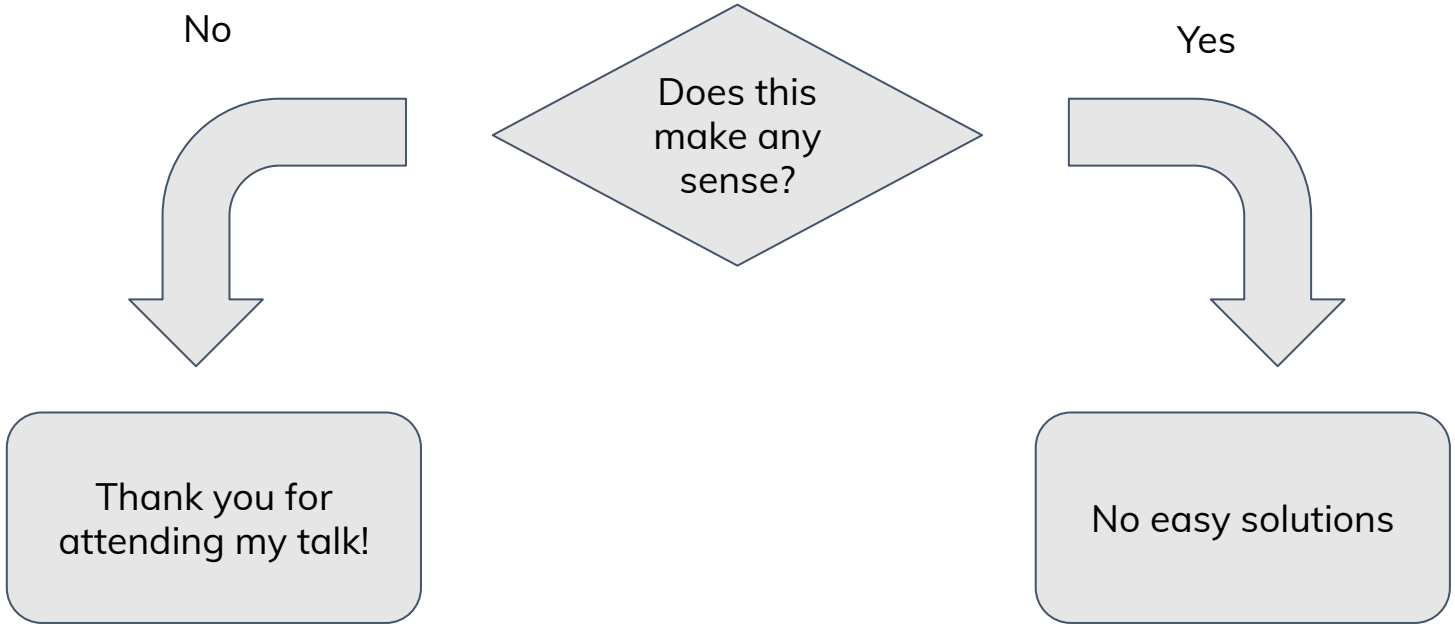




The way forward

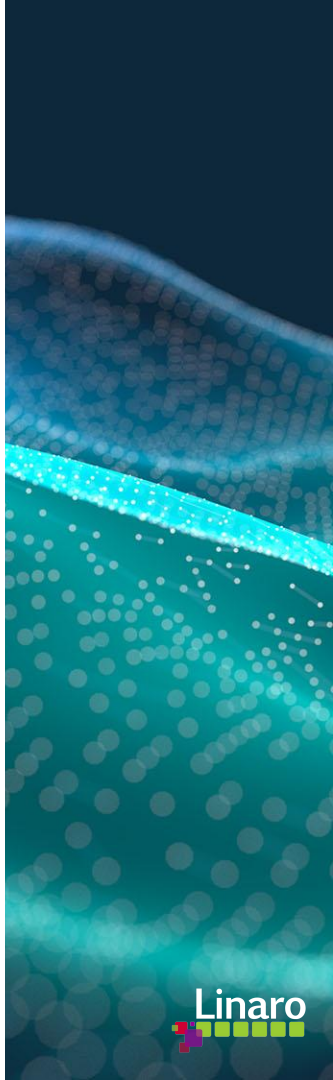


The way forward



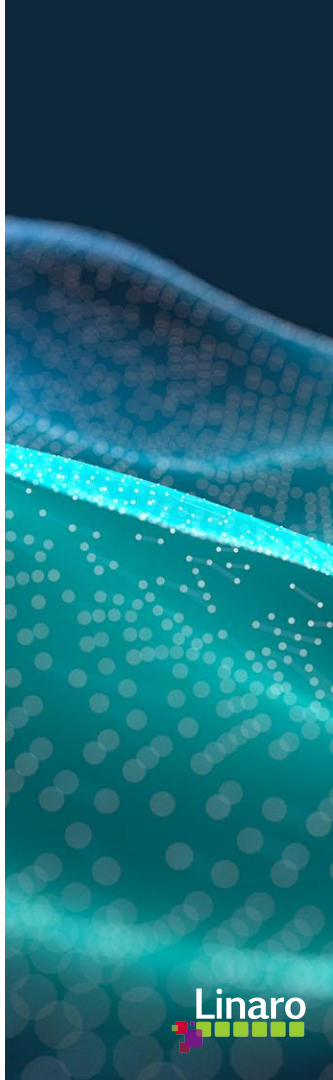
The way forward

- GPIO was mostly following the presented pattern but fixing issues still takes a lot of time
  - Lots of abuse by GPIOLIB users needs fixing before addressing fundamentals
- Complete overhaul of a subsystem would mean modifying tens of provider drivers
- Proposition: dedicated abstraction layer for resource providers
  - Abstract the notion of resource handles on the consumer side
  - Abstract the notion of logical devices for providers



Resource provider abstraction?

There's no problem in Computer Science that can't be solved by adding another layer of abstraction to it.



Resource provider abstraction?

```
struct gpio_desc *gpiod_get(...)
{
    return resmgr_get(&gpio_res, ...);
}

struct regulator *regulator_get(...)
{
    return resmgr_get(&regulator_res,
...);
}

int gpiod_get_value(...)
{
    guard(resmgr)(&desc->res);
    ...
}
```

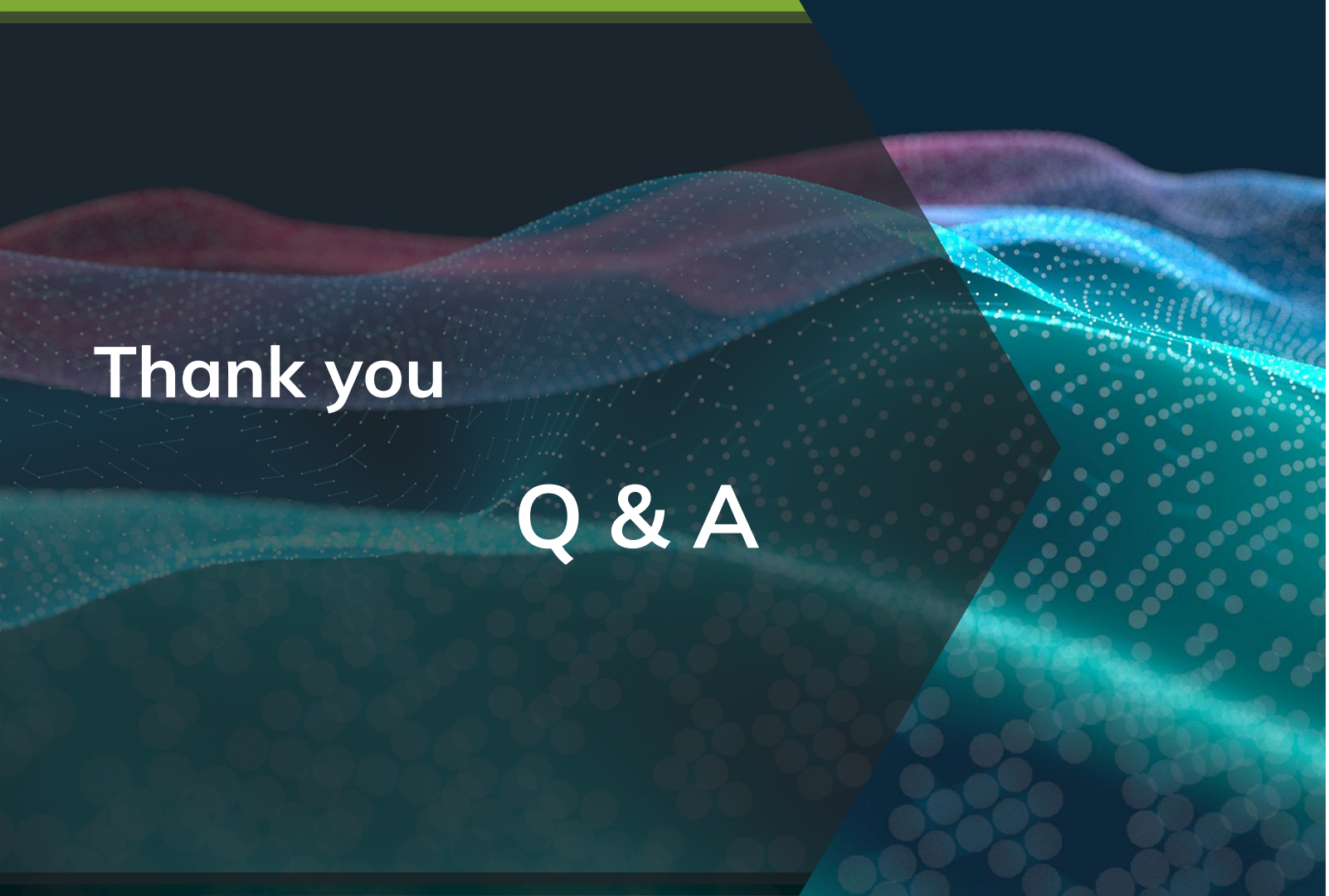
```
struct gpio device {
    struct resmgr_prvd shell;
    ...
};

struct resmgr shell {
    void __rcu *impl;
    ...
};
```

```
static
DEFINE_RESOURCE_PROVIDER(gpio_res);
```

```
int resmgr_add_provider(struct resmgr_prvd
*prvd);
int resmgr_del_provider(struct resmgr_prvd
*prvd);
```

- Take care of concurrent access
- Handle lookups
- Handle provider removal

Linaro



Linaro