



Keeping Track of Resources in the Kernel - Overview and Pitfalls of Resource Management Mechanisms

Bartosz Golaszewski

Senior Staff Engineer, Qualcomm

Prague, Open Source Summit 2026

Snapdragon and Qualcomm branded products are products of Qualcomm Technologies, Inc. and/or its subsidiaries.



About me



Linux Kernel Developer for Qualcomm

Over 16 years of professional experience
22 years of using linux



Kernel Maintainer

Maintainer of the GPIO and power sequencing subsystems, Qualcomm pinctrl and several others



Author of libgpiod

User-space part of the GPIO uAPI



Open-source contributor to many other projects

Zephyr, Yocto, busybox, buildroot, Qt, mender, etc.

Motivation



Rust4Linux



Sashiko and other LLMs



Countless driver code reviews



Previous work on object lifetime issues

C is a spartan language



Agenda



Devres



`cleanup.h`



Reference counting



Modules

Devres



Devres

```
int xyz_probe(struct device *dev)
{
    struct foo *f;
    struct bar *b;

    f = kzalloc(sizeof(*f), GFP_KERNEL);
    if (!f)
        return -ENOMEM;

    b = kzalloc(sizeof(*b), GFP_KERNEL);
    if (!b) {
        kfree(f);
        return -ENOMEM;
    }

    return 0;
}

void xyz_remove(struct device *dev)
{
    kfree(b);
    kfree(f);
}
```

Allows
going
from this

To this

```
int xyz_probe(struct device *dev)
{
    struct foo *f;
    struct bar *b;

    f = devm_kzalloc(dev, sizeof(*f), GFP_KERNEL);
    if (!f)
        return -ENODEV;

    b = devm_kzalloc(dev, sizeof(*b), GFP_KERNEL);
    if (!b)
        return -ENODEV;

    return 0;
}
```

Devres

Devres

- Has been around for ~20 years

Devres

- Has been around for ~20 years
- Aimed at simplifying driver development by outsourcing resource management to a higher abstraction layer

Devres

- Has been around for ~20 years
- Aimed at simplifying driver development by outsourcing resource management to a higher abstraction layer
- Tightly coupled with the lifetime of struct device

Devres

- Has been around for ~20 years
- Aimed at simplifying driver development by outsourcing resource management to a higher abstraction layer
- Tightly coupled with the lifetime of `struct device`
- Implemented as a linked-list of memory chunks with associated release callbacks

Devres

- Has been around for ~20 years
- Aimed at simplifying driver development by outsourcing resource management to a higher abstraction layer
- Tightly coupled with the lifetime of `struct device`
- Implemented as a linked-list of memory chunks with associated release callbacks
- Resources released in the reverse order of how they were acquired

Devres

Devres

- Problem: devres gives users the impression of not having to care about the underlying objects' lifetimes

Devres

- Problem: devres gives users the impression of not having to care about the underlying objects' lifetimes
 - Devres resources are really only designed to live from driver bind to unbind...

Devres

- Problem: devres gives users the impression of not having to care about the underlying objects' lifetimes
 - Devres resources are really only designed to live from driver bind to unbind...
 - ... but the API contract is not strict

Devres

- Problem: devres gives users the impression of not having to care about the underlying objects' lifetimes
 - Devres resources are really only designed to live from driver bind to unbind...
 - ... but the API contract is not strict
- Devices are reference counted, devres resources are not

Devres - lifetime of struct device

Devres - lifetime of struct device

`device_initialize()`

`device_add()`

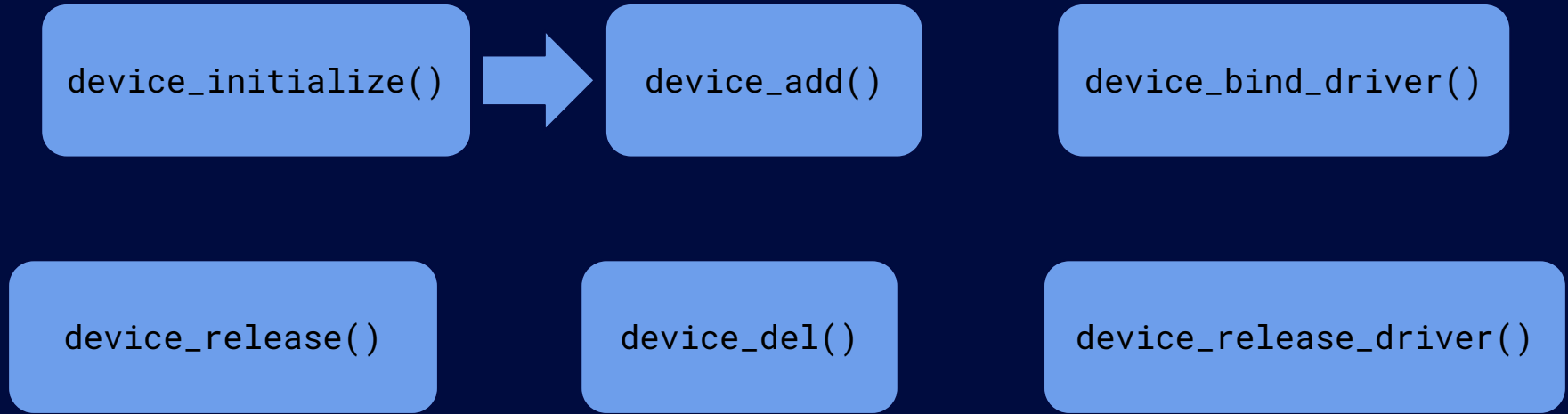
`device_bind_driver()`

`device_release()`

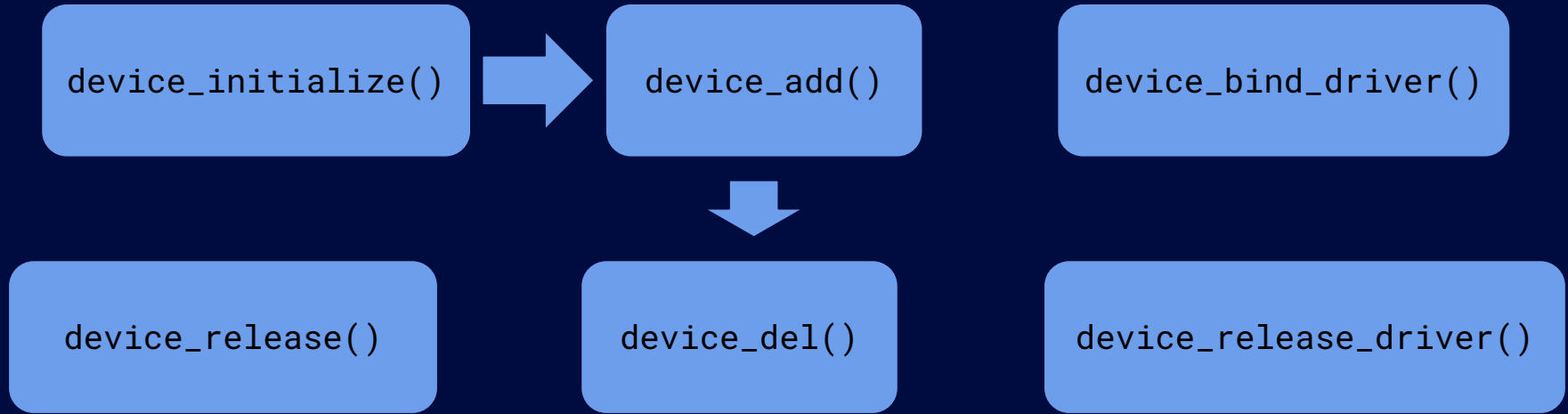
`device_del()`

`device_release_driver()`

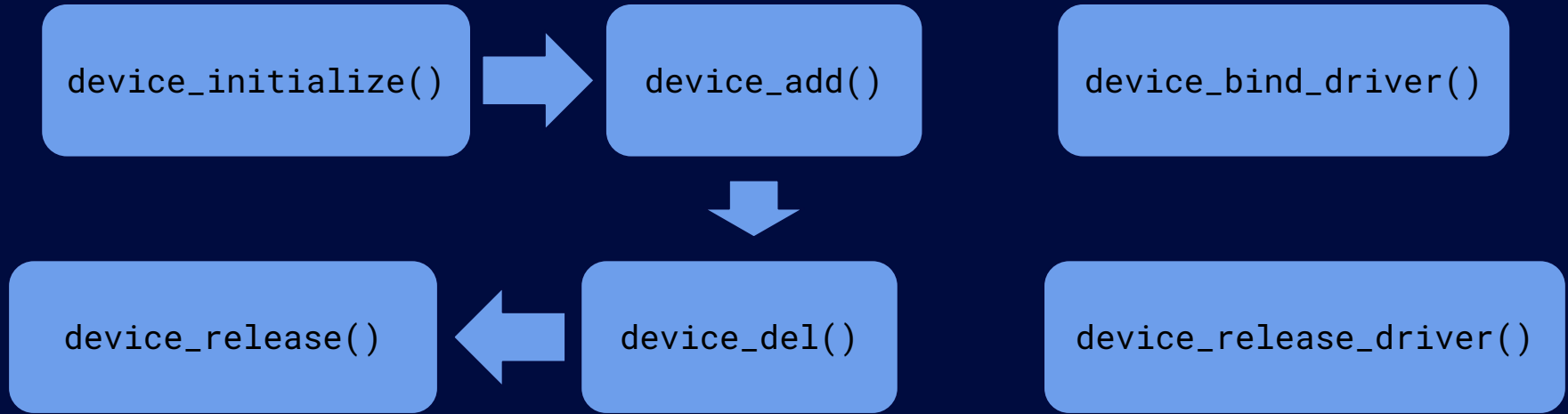
Devres - lifetime of struct device



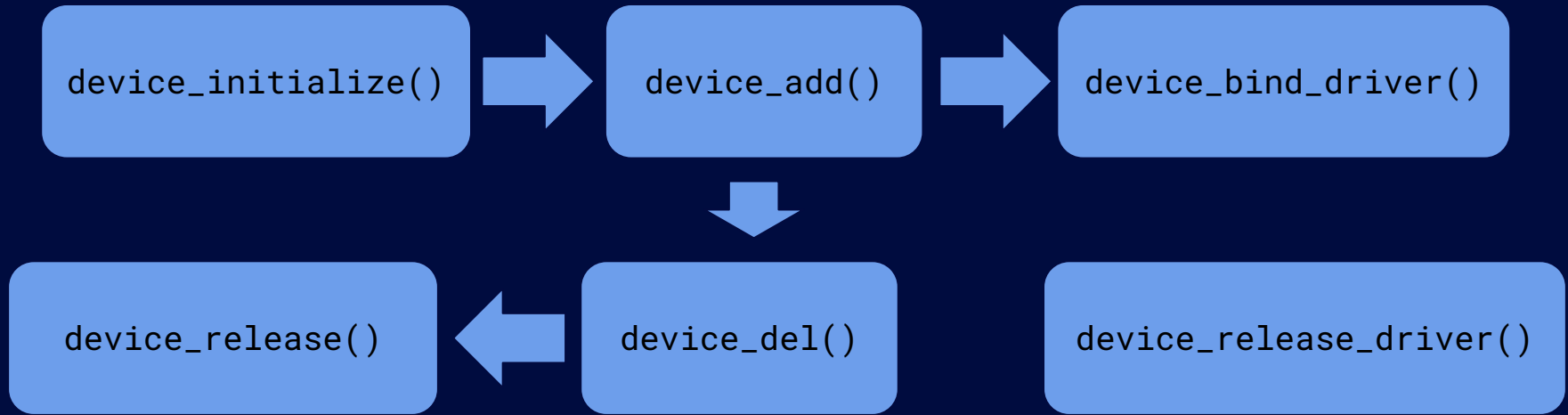
Devres - lifetime of struct device



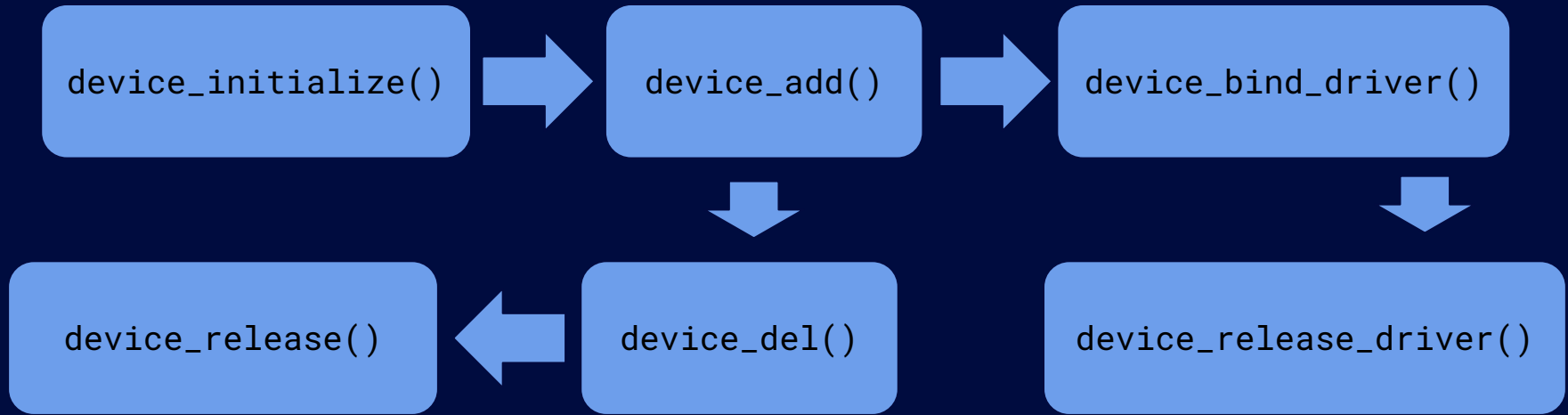
Devres - lifetime of struct device



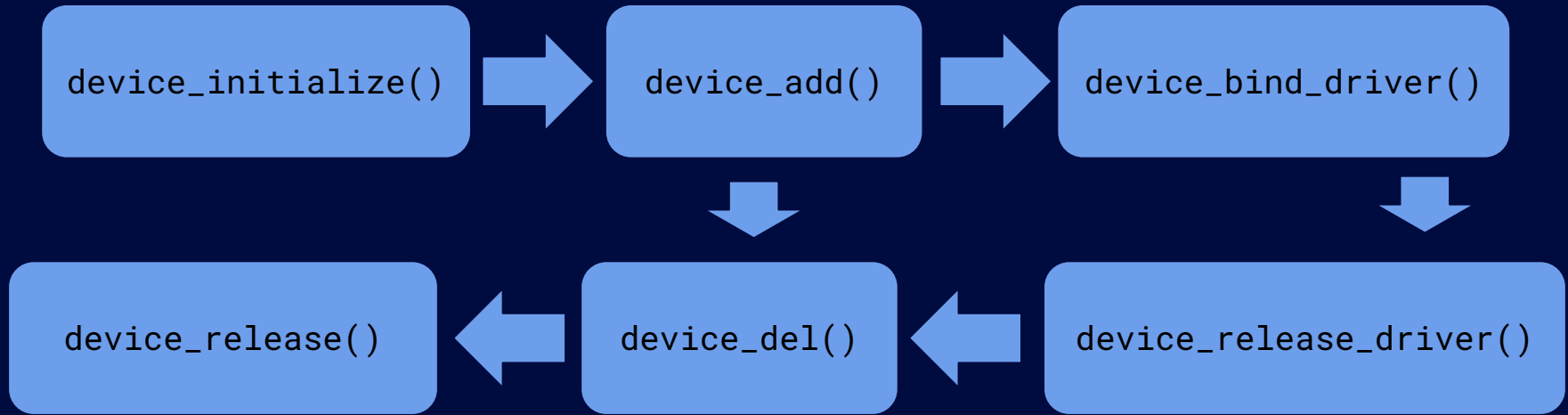
Devres - lifetime of struct device



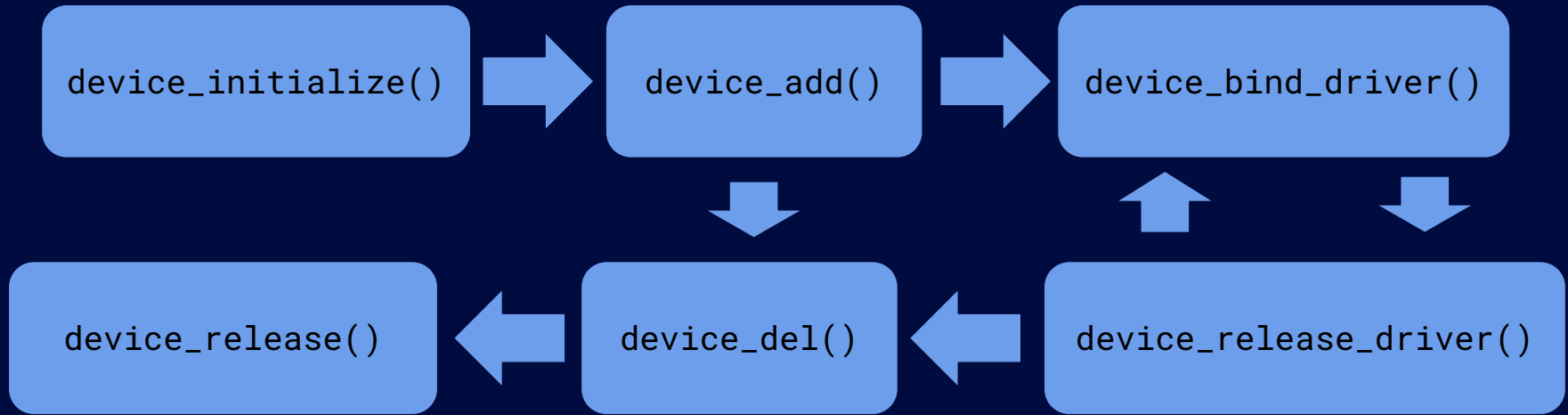
Devres - lifetime of struct device



Devres - lifetime of struct device

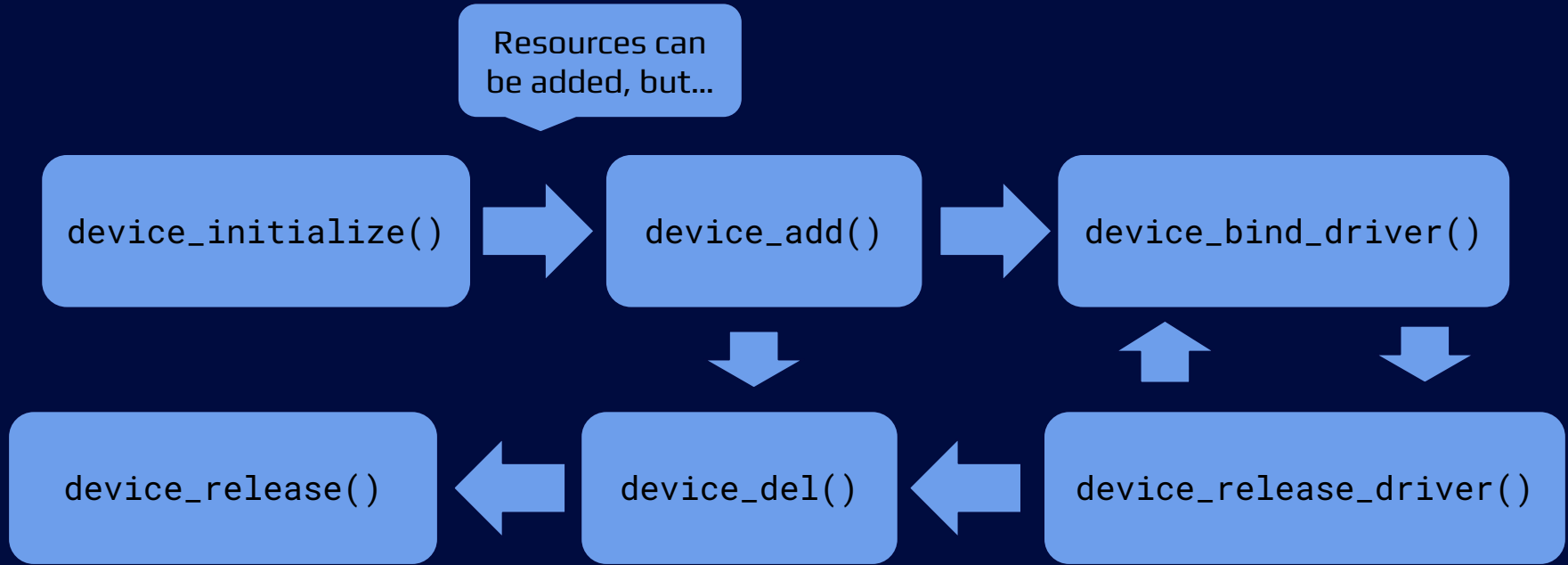


Devres - lifetime of struct device

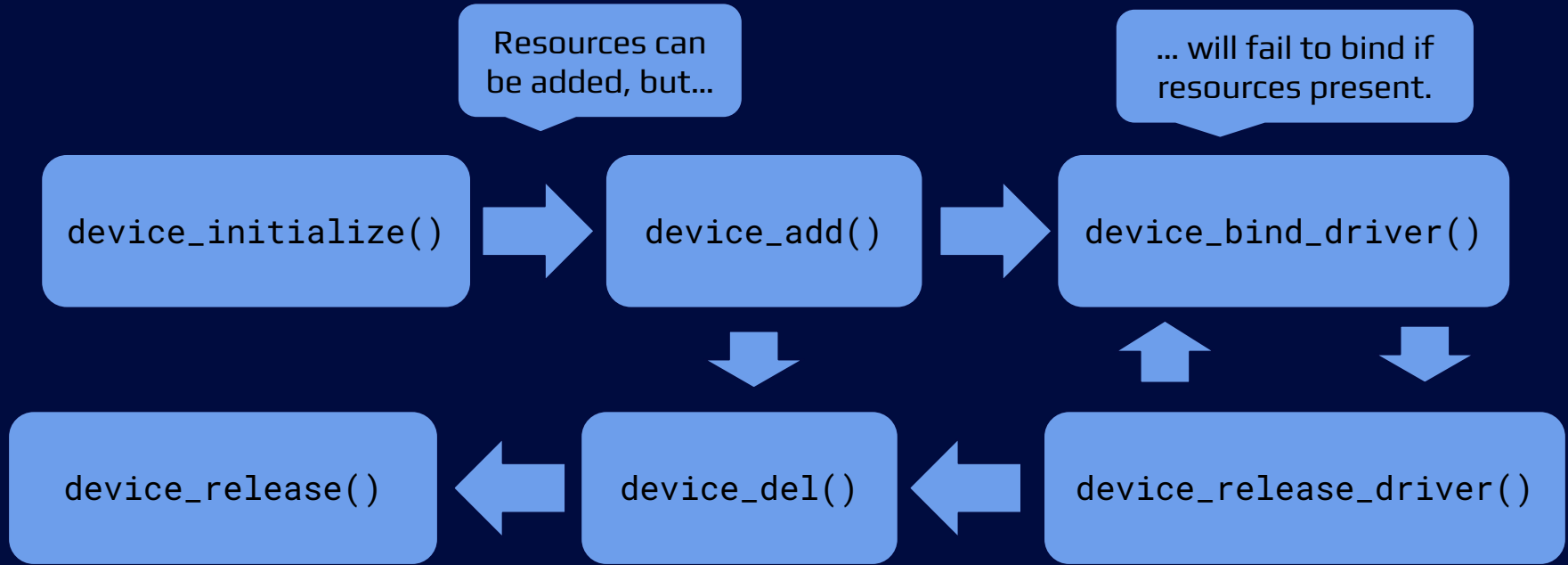


Devres - lifetime of struct device

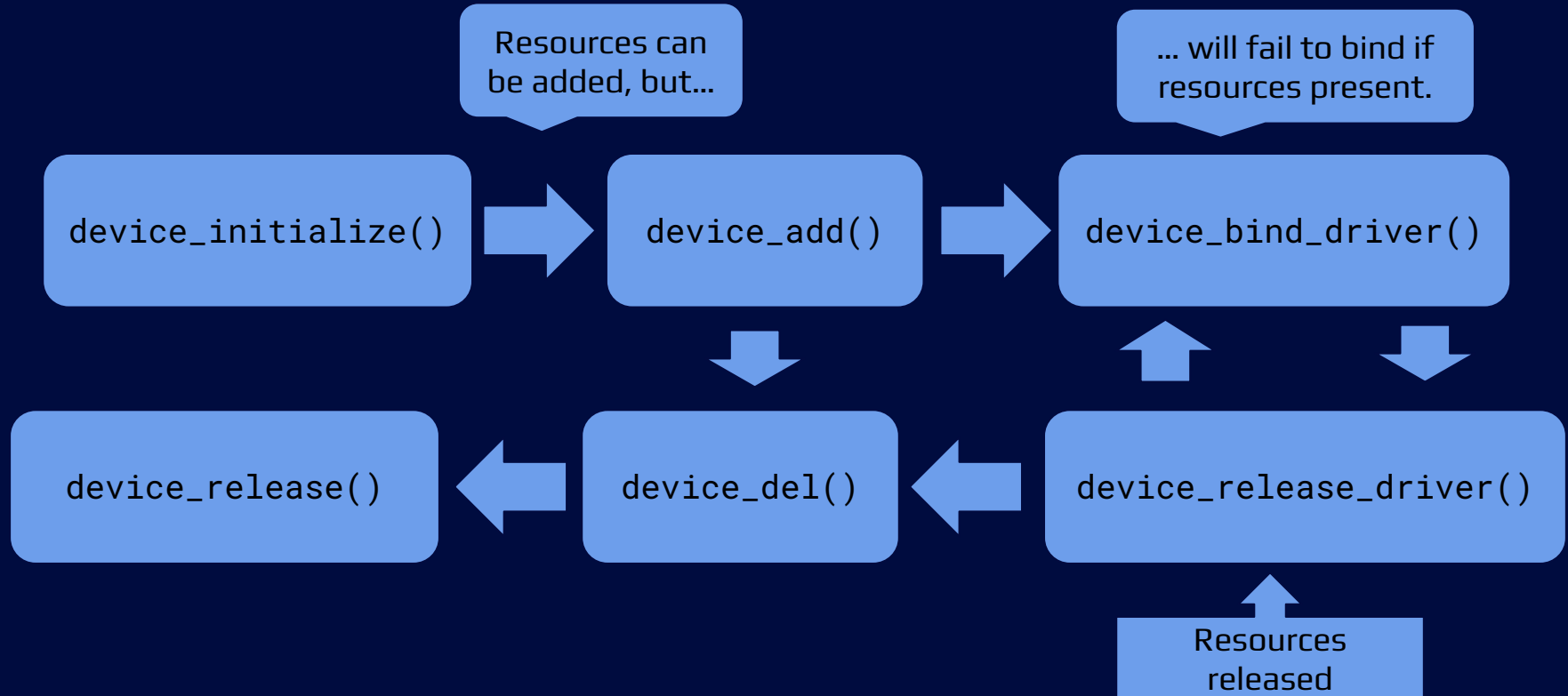
Resources can
be added, but...



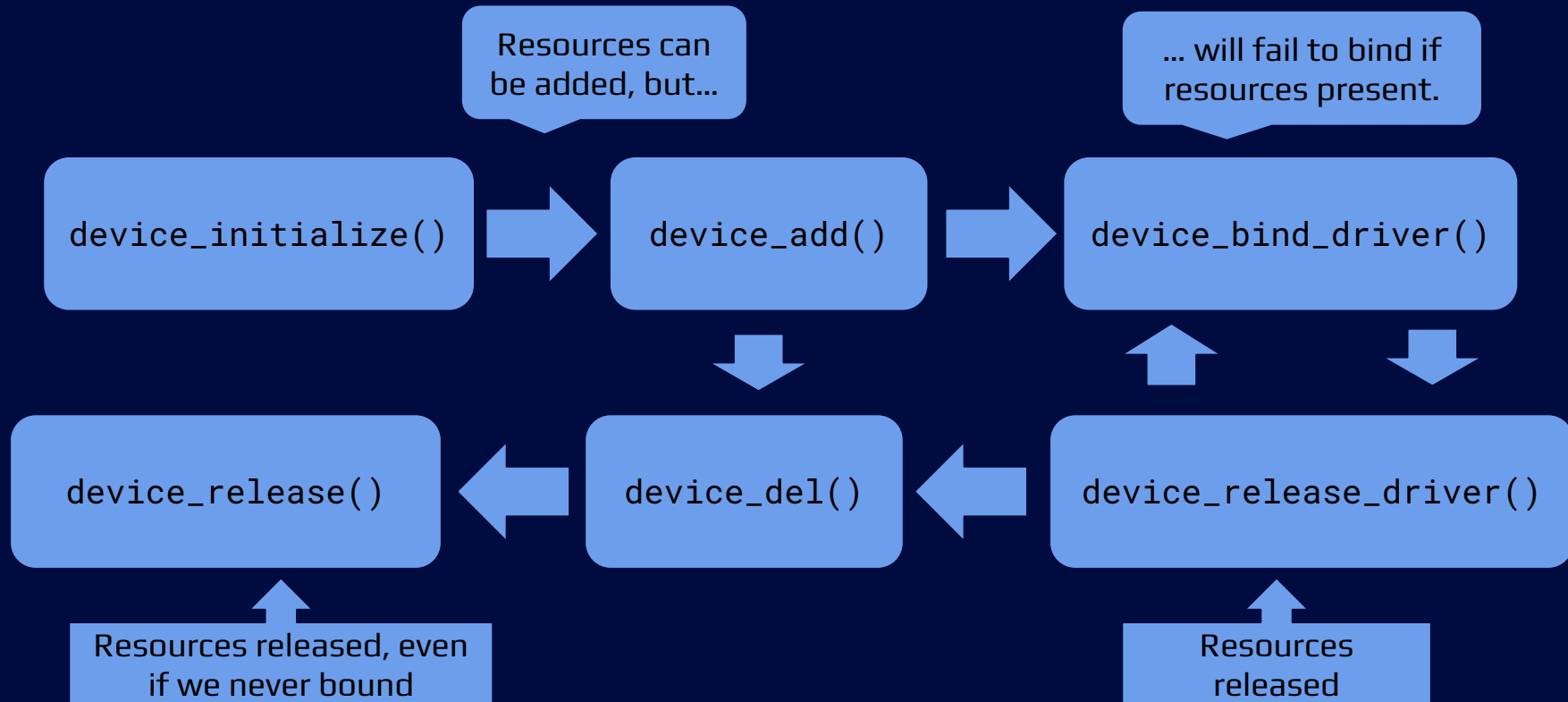
Devres - lifetime of struct device



Devres - lifetime of struct device



Devres - lifetime of struct device



Devres Dont's

Devres Dont's

- Don't use devres in sub-system code

Devres Dont's

```
469 static int nvmem_populate_sysfs_cells(struct nvmem_device *nvmem)
470 {
471     struct attribute_group group = {
472         .name = "cells",
473     };
474     struct nvmem_cell_entry *entry;
475     const struct bin_attribute **pattrs;
476     struct bin_attribute *attrs;
477     unsigned int ncells = 0, i = 0;
478     int ret;
479
480     guard(mutex)(&nvmem_mutex);
481
482     if (list_empty(&nvmem->cells) || nvmem->sysfs_cells_populated)
483         return 0;
484
485     /* Allocate an array of attributes with a sentinel */
486     ncells = list_count_nodes(&nvmem->cells);
487     pattrs = devm_kcalloc(&nvmem->dev, ncells + 1,
488                         sizeof(struct bin_attribute *), GFP_KERNEL);
489     if (!pattrs)
490         return -ENOMEM;
491
492     attrs = devm_kcalloc(&nvmem->dev, ncells, sizeof(struct bin_attribute), GFP_KERNEL);
493     if (!attrs)
494         return -ENOMEM;
```

Devres Dont's

NVMEM provider unbind

Devres Dont's

NVMEM provider unbind



sysfs attribute memory freed

Devres Dont's

NVMEM provider unbind



sysfs attribute memory freed



kn->priv still points to it

Devres Dont's

NVMEM provider unbind



sysfs attribute memory freed



kn->priv still points to it

Sysfs group only removed at
`kobject_del()`

Devres Dont's

- Don't use devres in sub-system code
- Don't add device resources for devices whose lifetime you don't control

Devres Dont's

```
> + config.stride = 1;
> + config.read_only = true;
> + config.root_only = true;
> + config.ignore_wp = true;
> + config.of_node = to_of_node(dev->fwnode);
> +
> + bnv->devt = bdev->bd_device.devt;
> + bnv->nvmem = nvmem_register(&config);
> + if (IS_ERR(bnv->nvmem)) {
> +     dev_err_probe(&bdev->bd_device, PTR_ERR(bnv->nvmem),
> +                  "Failed to register NVMEM device\n");
> +
> +     kfree(bnv);
> +     return PTR_ERR(bnv->nvmem);
> + }
> +
> + /*
> +  * Free bnv only when the nvmem device is fully released (i.e. when
> +  * its kref hits zero), not at unregister time. This prevents a
> +  * use-after-free if a consumer still holds an nvmem_cell reference
> +  * when the block device is removed: nvmem_unregister() only does a
> +  * kref_put(), so reg_read could still be called with bnv as priv
> +  * until the last consumer drops its cell.
> +  */
> + if (devm_add_action(nvmem_dev(bnv->nvmem), blk_nvmem_free, bnv)) {
> +     nvmem_unregister(bnv->nvmem);
> +     kfree(bnv);
> +     return -ENOMEM;
> + }
```

Devres Dont's

```
> + config.stride = 1;
> + config.read_only = true;
> + config.root_only = true;
> + config.ignore_wp = true;
> + config.of_node = to_of_node(dev->fwnode);
> +
> + bnv->devt = bdev->bd_device.devt;
> + bnv->nvmem = nvmem_register(&config);
> + if (IS_ERR(bnv->nvmem)) {
> +     dev_err_probe(&bdev->bd_device, PTR_ERR(bnv->nvmem),
> +                  "Failed to register\n");
> +     kfree(bnv);
> +     return PTR_ERR(bnv->nvmem);
> + }
> +
> + /*
> +  * Free bnv only when the nvmem device is fully released (i.e. when
> +  * its kref hits zero), not at unregister time. This prevents a
> +  * use-after-free if a consumer still holds an nvmem_cell reference
> +  * when the block device is removed: nvmem_unregister() only does a
> +  * kref_put(), so reg_read could still be called with bnv as priv
> +  * until the last consumer drops its cell.
> +  */
> + if (devm_add_action(nvmem_dev(bnv->nvmem), blk_nvmem_free, bnv)) {
> +     nvmem_unregister(bnv->nvmem);
> +     kfree(bnv);
> +     return -ENOMEM;
> + }
```

Comment making an assumption for which there's no API contract

Devres Dont's

```
1655 static int npcm_video_ece_init(struct npcm_video *video)
1656 {
1657     struct device_node *ece_node __free(device_node) = NULL;
1658     struct device *dev = video->dev;
1659     struct platform_device *ece_pdev;
1660     void __iomem *regs;
1661
1662     ece_node = of_parse_phandle(video->dev->of_node, "nuvoton,ece", 0);
1663     if (!ece_node) {
1664         dev_err(dev, "Failed to get ECE phandle in DTS\n");
1665         return -ENODEV;
1666     }
1667
1668     video->ece.enable = of_device_is_available(ece_node);
1669
1670     if (video->ece.enable) {
1671         dev_info(dev, "Support HEXTILE pixel format\n");
1672
1673         ece_pdev = of_find_device_by_node(ece_node);
1674         if (!ece_pdev) {
1675             dev_err(dev, "Failed to find ECE device\n");
1676             return -ENODEV;
1677         }
1678         struct device *ece_dev __free(put_device) = &ece_pdev->dev;
1679
1680         regs = devm_platform_ioremap_resource(ece_pdev, 0);
1681         if (IS_ERR(regs)) {
1682             dev_err(dev, "Failed to parse ECE reg in DTS\n");
1683             return PTR_ERR(regs);
1684         }
1685
1686         video->ece.regmap = devm_regmap_init_mmio(dev, regs,
1687             &npcm_video_ece_regmap_cfg);
1688         if (IS_ERR(video->ece.regmap)) {
1689             dev_err(dev, "Failed to initialize ECE regmap\n");
1690             return PTR_ERR(video->ece.regmap);
1691         }
1692
1693         video->ece.reset = devm_reset_control_get(ece_dev, NULL);
1694         if (IS_ERR(video->ece.reset)) {
1695             dev_err(dev, "Failed to get ECE reset control in DTS\n");
1696             return PTR_ERR(video->ece.reset);
1697         }
1698     }
```

Devres Dont's

```
1655 static int npcm_video_ece_init(struct npcm_video *video)
1656 {
1657     struct device_node *ece_node __free(device_node) = NULL;
1658     struct device *dev = video->dev;
1659     struct platform_device *ece_pdev;
1660     void __iomem *regs;
1661
1662     ece_node = of_parse_phandle(video->dev->of_node, "nuvoton,ece", 0);
1663     if (!ece_node) {
1664         dev_err(dev, "Failed to get ECE phandle in DTS\n");
1665         return -ENODEV;
1666     }
1667
1668     video->ece.enable = of_device_is_available(ece_node);
1669
1670     if (video->ece.enable) {
1671         dev_info(dev, "Support HEXTILE pixel format\n");
1672
1673         ece_pdev = of_find_device_by_node(ece_node);
1674         if (!ece_pdev) {
1675             dev_err(dev, "Failed to find ECE device\n");
1676             return -ENODEV;
1677         }
1678         struct device *ece_dev __free(put_device) = &ece_pdev->dev;
1679
1680         regs = devm_platform_ioremap_resource(ece_pdev, 0);
1681         if (IS_ERR(regs)) {
1682             dev_err(dev, "Failed to parse ECE reg in DTS\n");
1683             return PTR_ERR(regs);
1684         }
1685
1686         video->ece.regmap = devm_regmap_init_mmio(dev, regs,
1687             &npcm_video_ece_regmap_cfg);
1688         if (IS_ERR(video->ece.regmap)) {
1689             dev_err(dev, "Failed to initialize ECE regmap\n");
1690             return PTR_ERR(video->ece.regmap);
1691         }
1692
1693         video->ece.reset = devm_reset_control_get(ece_dev, NULL);
1694         if (IS_ERR(video->ece.reset)) {
1695             dev_err(dev, "Failed to get ECE reset control in DTS\n");
1696             return PTR_ERR(video->ece.reset);
1697         }
1698     }
```

ece_pdev may not be bound to a driver, its resources may not be freed until it's released; reprobing of npcm_video will fail

Devres Dont's

- Don't use devres in sub-system code
- Don't add device resources for devices whose lifetime you don't control
- Don't use devres for resources whose ownership is passed to sub-systems

Devres Dont's

```
case MAX77621:
    /*
     * We do not let the core switch this regulator on/off,
     * we just leave it on.
     */
    gpod = devm_gpod_get_optional(&client->dev,
                                "maxim,enable",
                                GPIOD_OUT_HIGH);

    if (IS_ERR(gpod))
        return PTR_ERR(gpod);
    if (gpod)
        max->enable_external_control = true;

    max->desc.enable_reg = MAX8973_VOUT;
    max->desc.enable_mask = MAX8973_VOUT_ENABLE;
    max->ops.enable = regulator_enable_regmap;
    max->ops.disable = regulator_disable_regmap;
    max->ops.is_enabled = regulator_is_enabled_regmap;
    max->ops.set_current_limit = max8973_set_current_limit;
    max->ops.get_current_limit = max8973_get_current_limit;
    break;
default:
    break;
}

ret = max8973_init_dcdc(max, pdata);
if (ret < 0) {
    dev_err(max->dev, "Max8973 Init failed, err = %d\n", ret);
    return ret;
}

config.dev = &client->dev;
config.init_data = pdata->reg_init_data;
config.driver_data = max;
config.of_node = client->dev.of_node;
config.regmap = max->regmap;

/*
 * Register the regulators
 * Turn the GPIO descriptor over to the regulator core for
 * lifecycle management if we pass an ena_gpod.
 */
if (config.ena_gpod)
    devm_gpod_unhinge(&client->dev, config.ena_gpod);
rdev = devm_regulator_register(&client->dev, &max->desc, &config);
if (IS_ERR(rdev)) {
    ret = PTR_ERR(rdev);
    dev_err(max->dev, "regulator register failed, err %d\n", ret);
    return ret;
}
```

Devres Do's

Devres Do's

- Only add resources at probe () time in device drivers

Devres Do's

- Only add resources at `probe()` time in device drivers
- Use the right `struct device`
 - The one bound to the driver allocating resources

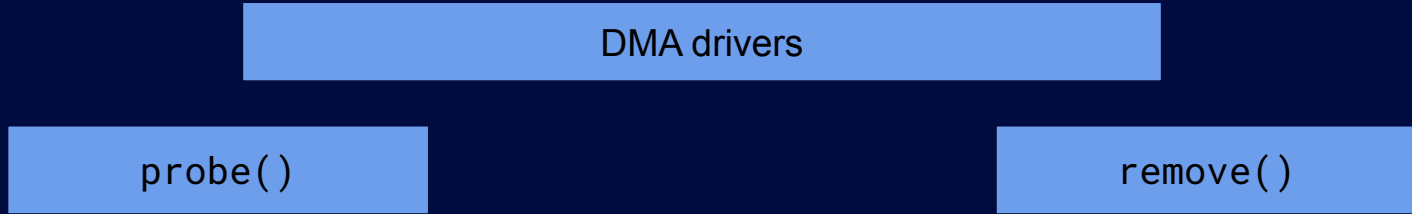
Devres Do's

- Only add resources at `probe()` time in device drivers
- Use the right `struct device`
 - The one bound to the driver allocating resources
- Pay attention to ordering at `remove()` time

Devres Do's

DMA drivers

Devres Do's



Devres Do's

DMA drivers

probe()

remove()

INIT_WORK()

clk_prepare_enable()

request_irq()

dma_async_device_register()

Devres Do's

DMA drivers

`probe()`

`INIT_WORK()`

`clk_prepare_enable()`

`request_irq()`

`dma_async_device_register()`

`remove()`

`dma_async_device_unregister()`

`free_irq()`

`cancel_work_sync()`

`clk_disable_unprepare()`

cleanup.h



`cleanup.h` (poor man's RAII in C)



cleanup.h

- Scope-based cleanup helpers

cleanup.h

- Scope-based cleanup helpers
- Automatically perform an action when a variable goes out of scope

cleanup.h

- Scope-based cleanup helpers
- Automatically perform an action when a variable goes out of scope
- Based on `__attribute__((cleanup(...)))`

cleanup.h



cleanup.h



`__free()`

- Simple cleanup for a single variable
- `__free(fn)` runs function `fn` when the annotated variable goes out of scope
- Use `no_free_ptr()` to clear a managed pointer
- Popularized by GLib (`g_auto()`, `g_autoptr()`, etc.)

cleanup.h



`__free()`

- Simple cleanup for a single variable
- `__free(fn)` runs function `fn` when the annotated variable goes out of scope
- Use `no_free_ptr()` to clear a managed pointer
- Popularized by GLib (`g_auto()`, `g_autoptr()`, etc.)



`CLASS()`

- Constructor/destructor pair for a complex type
- Handy for complex types used in loops



cleanup.h



`__free()`

- Simple cleanup for a single variable
- `__free(fn)` runs function `fn` when the annotated variable goes out of scope
- Use `no_free_ptr()` to clear a managed pointer
- Popularized by GLib (`g_auto()`, `g_autoptr()`, etc.)



`CLASS()`

- Constructor/destructor pair for a complex type
- Handy for complex types used in loops



`guard()` \
`scoped_guard()`

- Shorthand for a class with no need to reference the variable later within the scope
- In practice: mostly used for automatic releasing of locks

__free()

```
struct foo *make_foo(void)
{
    struct foo *f;
    int ret;

    f = kzalloc(sizeof(*f), GFP_KERNEL);
    if (!f)
        return -ENOMEM;

    ret = init_foo(f);
    if (ret) {
        kfree(f);
        return ret;
    }

    ret = init_foo_more(f);
    if (ret) {
        kfree(f);
        return ret;
    }

    return f;
}
```

Allows
going
from this

To this

```
struct foo *make_foo(void)
{
    int ret;

    struct foo *f __free(kfree) = kzalloc(sizeof(*f), GFP_KERNEL);
    if (!f)
        return -ENOMEM;

    ret = init_foo(f);
    if (ret)
        return ret;

    ret = init_foo_more(f);
    if (ret)
        return ret;

    return no_free_ptr(f);
}
```

__free()

Initialize managed variables at declaration

Do:

```
struct foo *f __free(kfree) = kzalloc(sizeof(*f), GFP_KERNEL);
```

Don't:

```
struct foo *f __free(kfree) = NULL;  
  
...  
  
f = kzalloc(sizeof(*f), GFP_KERNEL);
```

__free()

Don't mix goto and __free():

```
int do_xyz()
{
    int ret = 0;

    ...

    ret = do_something();
    if (ret)
        goto out;

    struct bar *b __free(kfree) = kzalloc(sizeof(*b), GFP_KERNEL);
    if (b)
        goto out;

out:
    return ret;
}
```

CLASS()

```
struct pwr {
    struct clk *clk;
    struct regulator *supply;
    struct icc_path *bw;
};

int pwr_enable(struct pwr *pwr)
{
    int ret;

    ret = clk_prepare_enable(pwr->clk);
    if (ret)
        return ret;

    ret = regulator_enable(pwr->supply);
    if (ret) {
        clk_disable_unprepare(pwr->clk);
        return ret;
    }

    ret = icc_set_bw(pwr->bw, FOO, BAR);
    if (ret) {
        regulator_disable(pwr->supply);
        clk_disable_unprepare(pwr->clk);
        return ret;
    }

    return 0;
}

void pwr_disable(struct pwr *pwr)
{
    icc_set_bw(pwr->bw, 0, 0);
    regulator_disable(pwr->supply);
    clk_disable_unprepare(pwr->clk);
}
```

Package as

```
DEFINE_CLASS(pwr_guard,
             int,
             pwr_enable(pwr),
             pwr_disable(pwr),
             struct pwr *pwr)

int do_something(struct pwr *pwr)
{
    CLASS(pwr_guard, guard)(pwr);
    if (guard)
        return guard;

    do_the_stuff();
    so_other_stuff();
}
```

guard() / scoped_guard()

```
int do_concurrently(void)
{
    int ret;

    mutex_lock(&mutex);

    ret = do_foo();
    if (ret) {
        mutex_unlock(&mutex);
        return ret;
    }

    ret = do_bar();
    if (ret) {
        mutex_unlock(&mutex);
        return ret;
    }

    mutex_unlock(&mutex);

    return ret;
}
```

Allows
going
from this

```
int do_concurrently(void)
{
    int ret;

    mutex_lock(&mutex);

    ret = do_foo();
    if (ret)
        goto out;

    ret = do_bar();
    if (ret)
        goto out;

out:
    mutex_unlock(&mutex);
    return ret;
}
```

To this

```
int go_concurrently(void)
{
    int ret;

    guard(mutex)(&mutex);

    ret = do_foo();
    if (ret)
        return ret;

    ret = do_bar();
    if (ret)
        return ret;

    return ret;
}
```

Or this

guard()

- guard() is effectively just a CLASS() you can't reference

scoped_guard()

- Implemented as a for loop
- “Watch your scope” - David Lechner:
<https://www.youtube.com/watch?v=GWttlIOLNgE>

scoped_guard()

```
int foo(void)
{
    for (int i = 0; i < XYZ; i++) {
        mutex_lock(&lock);

        if (should_stop(i))
            break;

        mutex_unlock(&lock);
    }
}
```



```
int foo(void)
{
    for (int i = 0; i < XYZ; i++) {
        scoped_guard(mutex, &lock) {
            if (should_stop(i))
                break;
        }
    }
}
```

scoped_guard()

Don't:

```
int foo(void)
{
    int x;

    scoped_guard(mutex, &lock) {
        x = get_x_locked();
    }

    return x;
}
```

Do:

```
int foo(void)
{
    int x = 0;

    scoped_guard(mutex, &lock) {
        x = get_x_locked();
    }

    return x;
}
```

Reference counting



Reference counting

Reference counting

- Lifecycle:
 - init
 - get
 - put
 - release

Reference counting

- Lifecycle:
 - init
 - get
 - put
 - release
- On **release** a custom function is called

Reference counting

- Lifecycle:
 - init
 - get
 - put
 - release
- On **release** a custom function is called
- Often based on atomically tracked struct `kref`

Reference counting

- Lifecycle:
 - init
 - get
 - put
 - release
- On **release** a custom function is called
- Often based on atomically tracked struct `kref`
- Works well together with `__free()/CLASS()`

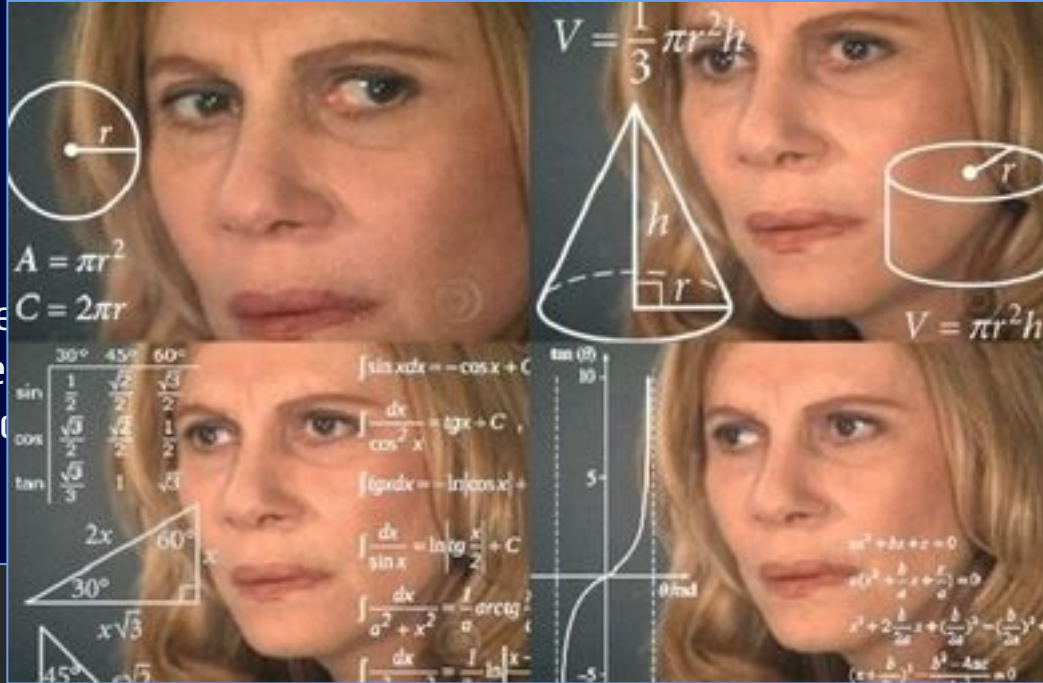
Reference counting

- Lifecycle:
 - init
 - get
 - put
 - release
- On **release** a custom function is called
- Often based on atomically tracked struct kref
- Works well together with __free()/CLASS()

```
struct device *dev __free(put_device) = bus_find_device_by_name( ... );
```

Reference counting

- Lifecycle:
 - init
 - get
 - put
 - release
- On *release
- Often based
- Works well



struct device

name(...);

Reference counting

```
commit e873c74132f0c5f1452816cd9bb26208f0bba1e1
```

```
Author: Shivank Garg <shivankg@amd.com>
```

```
Date: Sat Aug 22 19:22:05 2026 +0000
```

```
dmaengine: fix use-after-free in dma_chan_put() and dma_release_channel()
```

When `dma_device_put()` drops the last reference on `chan->device->ref`, `dma_device_release()` runs and may free the `dma_device` along with its channels.

`dma_chan_put()` then still reads `chan->device->owner` via `dma_chan_to_owner()` for the trailing `module_put()`. KASAN catches it:

```
slab-use-after-free in dma_chan_put+0x3e6/0x4c0
```

```
Read of size 8 by task insmod/6319
```

```
Freed by task 6319:
```

```
kfree+0x225/0x470
```

```
dma_chan_put+0x395/0x4c0
```

```
dmaengine_put+0xf8/0x160
```

Reference counting

```
diff --git a/drivers/dma/dmaengine.c b/drivers/dma/dmaengine.c
index e380b801df735..872ae0d2ed95d 100644
--- a/drivers/dma/dmaengine.c
+++ b/drivers/dma/dmaengine.c
@@ -500,10 +500,13 @@ static int dma_chan_get(struct dma_chan *chan)
 */
static void dma_chan_put(struct dma_chan *chan)
{
+   struct module *owner;
+
   /* This channel is not in use, bail out */
   if (!chan->client_count)
       return;

+   owner = dma_chan_to_owner(chan);
   chan->client_count--;

   /* This channel is not in use anymore, free it */
@@ -523,7 +526,7 @@ static void dma_chan_put(struct dma_chan *chan)
   /* This channel is not in use anymore, drop the device ref */
   if (!chan->client_count)
       dma_device_put(chan->device);
-   module_put(dma_chan_to_owner(chan));
+   module_put(owner);
}

enum dma_status dma_sync_wait(struct dma_chan *chan, dma_cookie_t cookie)
```

Reference counting

```
commit 8ad342a863590b24ce77681b7e081363fb3333f7
```

```
Author: Logan Gunthorpe <logang@deltatee.com>
```

```
Date: Mon Dec 16 12:01:19 2019 -0700
```

`dmaengine: Add reference counting to dma_device struct`

Adding a reference count helps drivers to properly implement the unbind while in use case.

References are taken and put every time a channel is allocated or freed.

Once the final reference is put, the device is removed from the `dma_device_list` and a release callback function is called to signal the driver to free the memory.

Signed-off-by: Logan Gunthorpe <logang@deltatee.com>

Link: <https://lore.kernel.org/r/20191216190120.21374-5-logang@deltatee.com>

Signed-off-by: Vinod Koul <vkoul@kernel.org>

Reference counting

- Attention to the API contract

Reference counting

- Attention to the API contract

```
/* Returns new reference. */  
struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device, struct pci_dev *from);  
/* Doesn't return a new reference. */  
static inline struct pci_dev *pci_upstream_bridge(struct pci_dev *dev)
```

Reference counting

- Attention to the API contract

```
/* Returns new reference. */  
struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device, struct pci_dev *from);  
/* Doesn't return a new reference. */  
static inline struct pci_dev *pci_upstream_bridge(struct pci_dev *dev)
```

```
/* Returns new reference. */  
struct device_node *of_get_parent(const struct device_node *node);  
/* Returns new reference AND drops reference on node. */  
struct device_node *of_get_next_parent(struct device_node *node);
```

Reference counting

- Scoped loops for refcounted objects <3

Reference counting

- Scoped loops for refcounted objects <3

```
struct fwnode_handle *child;

fwnode_for_each_child_node(fwnode, child) {
    ...
}

fwnode_handle_put(child);
```

Reference counting

- Scoped loops for refcounted objects <3

```
struct fwnode_handle *child;

fwnode_for_each_child_node(fwnode, child) {
    ...
}

fwnode_handle_put(child);
```

```
struct fwnode_handle *child;

fwnode_for_each_child_node_scoped(fwnode, child) {
    ...
}
```

Reference counting

- Devres-refcount cocktail

Reference counting

- Devres-refcount cocktail

```
struct subsys_data {  
    struct device dev;  
    int a;  
};  
  
struct drv_data {  
    struct subsys_data subsys;  
    int b;  
};
```

Reference counting

- Devres-refcount cocktail

```
struct subsys_data {  
    struct device dev;  
    int a;  
};  
  
struct drv_data {  
    struct subsys_data subsys;  
    int b;  
};
```

```
static int subsys_foo_init(struct subsys_data *data)  
{  
    ...  
  
    device_initialize(&data->dev);  
  
    ...  
  
    return 0;  
}
```

Reference counting

- Devres-refcount cocktail

```
static int foo_probe(struct device *dev)
{
    struct drv_data *data;
    int ret;

    data = devm_kzalloc(dev, sizeof(*data), GFP_KERNEL);
    if (!data)
        return -ENOMEM;

    ...

    ret = subsys_foo_init(&data->subsys);
    if (ret)
        return ret;

    ...

    return 0;
}
```

```
struct subsys_data {
    struct device dev;
    int a;
};

struct drv_data {
    struct subsys_data subsys;
    int b;
};
```

```
static int subsys_foo_init(struct subsys_data *data)
{
    ...

    device_initialize(&data->dev);

    ...

    return 0;
}
```

Reference counting

```
void devm_gpiod_unhinge(struct device *dev, struct gpio_desc *desc)
```

Reference counting

Do

- Refer to the kerneldocs for the refcounting API contract
- Use scoped loop variants

Don't

- Use after free (duh...)
- Don't mix refcounted and device-managed resources

Modules



Modules

- Special case of reference counting
- Users of module code must pin the module first and prevent it from unloading

```
struct module *owner;  
  
bool try_module_get(struct module *module);  
void module_put(struct module *module);
```

Modules

Provider

Subsystem

Consumer

Modules

Provider

Subsystem

Consumer

`subsys_register()`

Modules

Provider

`subsys_register()`

Subsystem

`remember_provider()`

Consumer

Modules

Provider

`subsys_register()`

Subsystem

`remember_provider()`

Consumer

`subsys_get()`

Modules

Provider

`subsys_register()`

Subsystem

`remember_provider()`

`try_module_get()`

Consumer

`subsys_get()`

Modules

Provider

`subsys_register()`

Subsystem

`remember_provider()`

`try_module_get()`

Consumer

`subsys_get()`

`provider_api()`

Modules

Provider

`subsys_register()`

Subsystem

`remember_provider()`

`try_module_get()`

Consumer

`subsys_get()`

`provider_api()`

`subsys_put()`

Modules

Provider

`subsys_register()`

Subsystem

`remember_provider()`

`try_module_get()`

`module_put()`

Consumer

`subsys_get()`

`provider_api()`

`subsys_put()`

Thank you

Nothing in these materials is an offer to sell any of the components or devices referenced herein.

© Qualcomm Technologies, Inc. and/or its affiliated companies. All Rights Reserved.

Qualcomm and Snapdragon are trademarks or registered trademarks of Qualcomm Incorporated. Other products and brand names may be trademarks or registered trademarks of their respective owners.

References in this presentation to “Qualcomm” may mean Qualcomm Incorporated, Qualcomm Technologies, Inc., and/or other subsidiaries or business units within the Qualcomm corporate structure, as applicable. Qualcomm Incorporated includes our licensing business, QTL, and the vast majority of our patent portfolio. Qualcomm Technologies, Inc., a subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of our engineering, research and development functions, and substantially all of our products and services businesses, including our QCT semiconductor business.

Snapdragon and Qualcomm branded products are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patents are licensed by Qualcomm Incorporated.

Follow us on: [in](#) [X](#) [@](#) [v](#) [f](#)

For more information, visit us at [qualcomm.com](https://www.qualcomm.com) & [qualcomm.com/blog](https://www.qualcomm.com/blog)

