The Right Way: Managed Resource Allocation in Linux Device Drivers

Eli Billauer

May 26th, 2014

PUBLIC DOMAIN

This work is released under Creative Common's CC0 license version 1.0 or later. To the extent possible under law, the author has waived all copyright and related or neighboring rights to this work.

Introduction

The life cycle of a device driver

- init (leading to registration of the driver)
- probe
- remove
- exit (if module is removed)

A driver's resources

- Memory for private data structures
- IRQs
- Memory region allocation (`request_mem_region()`)
- I/O mapping of memory regions (`ioremap()`)
- Buffer memory (possibly with DMA mapping)
- Esoterics: Clocks, GPIO, PWMs, USB phy, SPI masters, regulators, DMA controllers, etc.

Using the API

In short

The old way:

```
rc = request_irq(irq, my_isr, 0, my_name, my_data);

if (rc) {
  dev_err(dev, "Failed to register IRQ.\n");
  rc = -ENODEV;
  goto failed_register_irq; /* Unroll */
}
```

The right way:

```
rc = devm_request_irq(dev, irq, my_isr, 0,
                      my_name, my_data);

if (rc) {
  dev_err(dev, "Failed to register IRQ.\n");
  return -ENODEV; /* Automatic unroll */
}
```

Supported functions (from devres.txt)

- devm_kzalloc()
- devm_kfree()
- devm_kmemdup()
- devm_get_free_pages()
- devm_free_pages()
- devm_iio_device_alloc()
- devm_iio_device_free()
- devm_iio_trigger_alloc()
- devm_iio_trigger_free()
- devm_iio_device_register()
- devm_iio_device_unregister()
- devm_request_region()
- devm_request_mem_region()
- devm_release_region()
- devm_release_mem_region()
- devm_request_irq()
- devm_free_irq()
- dmam_alloc_coherent()
- dmam_free_coherent()
- dmam_alloc_noncoherent()
- dmam_free_noncoherent()
- dmam_declare_coherent_memory()
- dmam_pool_create()
- dmam_pool_destroy()
- dmam_map_single()
- dmam_unmap_single()
- dmam_map_single_attrs()
- dmam_unmap_single_attrs()
- devm_ioport_map()
- devm_ioport_unmap()
- devm_ioremap()
- devm_ioremap_nocache()
- devm_iounmap()
- devm_ioremap_resource()
- devm_request_and_ioremap()
- devm_acpi_dma_controller_register()
- devm_spi_register_master()
- pcim_enable_device()
- pcim_pin_device()
- pcim_map_single()
- pcim_unmap_single()
- pcim_iomap()
- pcim_iounmap()
- pcim_iomap_table()
- pcim_iomap_regions()
- devm_regulator_get()
- devm_regulator_put()
- devm_regulator_bulk_get()
- devm_regulator_register()
- devm_clk_get()
- devm_clk_put()
- devm_pinctrl_get()
- devm_pinctrl_put()
- devm_pwm_get()
- devm_pwm_put()
- devm_usb_get_phy()
- devm_usb_put_phy()

Why the old way is bad

- probe: If it fails in the middle, free anything allocated
- remove: Duplicate code of probe's error handling
- Resource leaks
- Oopses
- Hard to spot problems in failure handling

The kernel police is sleeping

- Managed resources was introduced in kernel 2.6.21 (2007) by Tejun Heo
- No rush to migrate drivers
- Not required in new drivers
- Many basic functions still missing (`__get_free_pages` anyone?)
- A good opportunity to get involved in the kernel development...?

Migrating to managed resources

In probe method:

- Look up the functions in Documentation/driver-model/devres.txt
- Migrate *all* resource allocation function calls
- Remove resource releases on error handling
- Replace goto's and other resource releases with a `return`.
- There are functions for "manually" freeing resources. Their need and API backward compatibility is questionable.

In remove method:

- Remove resource releases
- The method call is often reduced to almost nothing

Behind the scenes

- Each device structure ("dev") has a linked list of resources (`devres_head` in `struct device`).
- Calling an managed resource allocator involves adding the resource to the list.
- The resources are released in reverse order
    - when the probe method exits with an error status
    - after the remove method returns

How the remove method is called

From drivers/base/dd.c:

```
static void
 __device_release_driver(struct device *dev)
{
...
  if (dev->bus && dev->bus->remove)
    dev->bus->remove(dev);
   else if (drv->remove)
     drv->remove(dev);
  devres_release_all(dev);
  dev->driver = NULL;
  dev_set_drvdata(dev, NULL);
...
}
```

The guided tour

Let's look at some sources of drivers

Inserting callbacks into the release sequence

From drivers/input/touchscreen/auo-pixcir-ts.c:

```
static void auo_pixcir_reset(void *data)
{
  struct auo_pixcir_ts *ts = data;
  gpio_set_value(ts->pdata->gpio_rst, 0);
}
```

... and then in the probe method:

```
error = devm_add_action(&client->dev,
                        auo_pixcir_reset, ts);
if (error) {
  ...!!!... ;
  return error;
}
```

May the callback be time-consuming? Sleep?

Extra benefits

- devm_ioremap_resource(): devm_request_mem_region() and devm_ioremap() combined
- Same with pcim_iomap_regions()

Obtaining the first BAR of a PCI card:

```
rc = pcim_iomap_regions(pdev, 0x01, my_name);
if (rc) {
  dev_err(&pdev->dev, "!!!\n");
  return rc;
}

registers = pcim_iomap_table(pdev)[0];
```

Extra benefits (cont.)

pcim_enable_device() is useful because of its release function. drivers/pci/pci.c, in pcim_release():

```c
if (dev->msi_enabled)
  pci_disable_msi(dev);
if (dev->msix_enabled)
  pci_disable_msix(dev);

for (i = 0; i < DEVICE_COUNT_RESOURCE; i++)
  if (this->region_mask & (1 << i))
     pci_release_region(dev, i);

if (this->restore_intx)
  pci_intx(dev, this->orig_intx);

if (this->enabled && !this->pinned)
  pci_disable_device(dev);
```

Releasing intermediate resources

Only b and c will be released (error checks are missing, of course):

```c
void *mygroup;
a = devm_kzalloc(dev, sizeof(*a), GFP_KERNEL);

mygroup = devres_open_group(dev, NULL, GFP_KERNEL);
if (!mygroup)
  return -ENOMEM; /* OK in a probe method */
b = devm_kzalloc(dev, sizeof(*b), GFP_KERNEL);
c = devm_kzalloc(dev, sizeof(*c), GFP_KERNEL);
devres_close_group(dev, mygroup);
d = devm_kzalloc(dev, sizeof(*d), GFP_KERNEL);

devres_release_group(dev, mygroup);
e = devm_kzalloc(dev, sizeof(*e), GFP_KERNEL);
return 0;
```

“Making of”

Making of devm_get_free_pages

From drivers/base/devres.c (pending patch):

```
struct pages_devres {
  unsigned long addr;
  unsigned int order;
};

static void devm_pages_release(struct device *dev,
                               void *res)
{
  struct pages_devres *devres = res;

  free_pages(devres->addr, devres->order);
}
```

Making of devm_get_free_pages (cont.)

From drivers/base/devres.c, utilities (pending patch):

```
static int devm_pages_match(struct device *dev,
                            void *res, void *p)
{
        struct pages_devres *devres = res;
        struct pages_devres *target = p;

        return devres->addr == target->addr;
}
```

Note that `devres->order` is ignored.

Making of devm_get_free_pages (cont.)

From drivers/base/devres.c, the function itself (pending patch):

```
unsigned long
devm_get_free_pages(struct device *dev,
                    gfp_t gfp_mask,
                    unsigned int order)
{
  struct pages_devres *devres;
  unsigned long addr;

  addr = __get_free_pages(gfp_mask, order);
  if (unlikely(!addr))
    return 0;

  devres = devres_alloc(devm_pages_release,
         sizeof(struct pages_devres), GFP_KERNEL);
```

Making of devm_get_free_pages (cont.)

From drivers/base/devres.c, the function itself (pending patch):

```
  if (unlikely(!devres)) {
    free_pages(addr, order);
    return 0;
  }

  devres->addr = addr;
  devres->order = order;

  devres_add(dev, devres);
  return addr;
}
EXPORT_SYMBOL_GPL(devm_get_free_pages);
```

Making of devm_get_free_pages (cont.)

From drivers/base/devres.c, the “manual” free (pending patch):

```c
void devm_free_pages(struct device *dev,
                     unsigned long addr)
{
  struct pages_devres devres = { .addr = addr };

  WARN_ON(devres_release(dev, devm_pages_release,
          devm_pages_match, &devres));
}
EXPORT_SYMBOL_GPL(devm_free_pages);
```

- Oops, I broke the API: The “order” parameter isn’t required.
- Does it matter? Should this function be used ever? Use groups instead!

Making of devm_ioremap

From lib/devres.c, utility functions:

```
static void devm_ioremap_release(
                    struct device *dev, void *res)
{
  iounmap(*(void __iomem **)res);
}

static int devm_ioremap_match(struct device *dev,
                       void *res, void *match_data)
{
  return *(void **)res == match_data;
}
```

Making of devm_ioremap (cont.)

From lib/devres.c, the function itself:

```
void __iomem *devm_ioremap(struct device *dev,
      resource_size_t offset, unsigned long size)
{
  void __iomem **ptr, *addr;
  ptr = devres_alloc(devm_ioremap_release,
                     sizeof(*ptr), GFP_KERNEL);
  if (!ptr)
    return NULL;

  addr = ioremap(offset, size);
  if (addr) {
    *ptr = addr;
    devres_add(dev, ptr);
  } else
    devres_free(ptr);
  return addr;
}
```

Making of devm_ioremap (cont.)

From lib/devres.c, the manual release:

```
void devm_iounmap(struct device *dev,
                  void __iomem *addr)
{
  WARN_ON(devres_destroy(dev, devm_ioremap_release,
                         devm_ioremap_match,
                         (void *)addr));
  iounmap(addr);
}

EXPORT_SYMBOL(devm_ioremap);
EXPORT_SYMBOL(devm_iounmap);
```

- `devm_ioremap_release` is merely used as an matching identifier for ioremap() entries
- Call `devres_release()` instead of `devres_destroy()`, with the same arguments, and drop iounmap(addr)

Wrap-up

Sources

In the kernel's source tree:

- Documentation/driver-model/devres.txt
- drivers/base/devres.c
- lib/devres.c
- drivers/base/dma-mapping.c

Thank you!

Questions?