Devres - Managed Device Resource

Tejun Heo <teheo@suse.de>

First draft 10 January 2007

1. Intro

devres came up while trying to convert libata to use iomap. Each iomapped address should be kept and unmapped on driver detach. For example, a plain SFF ATA controller (that is, good old PCI IDE) in native mode makes use of 5 PCI BARs and all of them should be maintained.

As with many other device drivers, libata low level drivers have sufficient bugs in ->remove and ->probe failure path. Well, yes, that's probably because libata low level driver developers are lazy bunch, but aren't all low level driver developers? After spending a day fiddling with braindamaged hardware with no document or braindamaged document, if it's finally working, well, it's working.

For one reason or another, low level drivers don't receive as much attention or testing as core code, and bugs on driver detach or initialization failure don't happen often enough to be noticeable. Init failure path is worse because it's much less travelled while needs to handle multiple entry points.

So, many low level drivers end up leaking resources on driver detach and having half broken failure path implementation in ->probe() which would leak resources or even cause oops when failure occurs. iomap adds more to this mix. So do msi and msix.

2. Devres

devres is basically linked list of arbitrarily sized memory areas associated with a struct device. Each devres entry is associated with a release function. A devres can be released in several ways. No matter what, all devres entries are released on driver detach. On release, the associated release function is invoked and then the devres entry is freed.

Managed interface is created for resources commonly used by device drivers using devres. For example, coherent DMA memory is acquired using dma_alloc_coherent(). The managed version is called dmam_alloc_coherent(). It is identical to dma_alloc_coherent() except for the DMA memory allocated using it is managed and will be automatically released on driver detach. Implementation looks like the following:

```
struct dma_devres {
      size_t          size;
      void            *vaddr;
      dma_addr_t      dma_handle;
};

static void dmam_coherent_release(struct device *dev, void *res)
{
      struct dma_devres *this = res;

      dma_free_coherent(dev, this->size, this->vaddr, this->dma_handle);
}

dmam_alloc_coherent(dev, size, dma_handle, gfp)
{
      struct dma_devres *dr;
      void *vaddr;

      dr = devres_alloc(dmam_coherent_release, sizeof(*dr), gfp);
      ...

      /* alloc DMA memory as usual */
      vaddr = dma_alloc_coherent(...);
      ...

      /* record size, vaddr, dma_handle in dr */
      dr->vaddr = vaddr;
      ...

      devres_add(dev, dr);

      return vaddr;
}
```

If a driver uses dmam_alloc_coherent(), the area is guaranteed to be freed whether initialization fails half-way or the device gets detached. If most resources are acquired using managed interface, a driver can have much simpler init and exit code. Init path basically looks like the following:

```
my_init_one()
{
```

```
        struct mydev *d;

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

        d->ring = dmam_alloc_coherent(...);
        if (!d->ring)
                return -ENOMEM;

        if (check something)
                return -EINVAL;
        ...

        return register_to_upper_layer(d);
  }
```

And exit path:

```
  my_remove_one()
  {
        unregister_from_upper_layer(d);
        shutdown_my_hardware();
  }
```

As shown above, low level drivers can be simplified a lot by using devres. Complexity is shifted from less maintained low level drivers to better maintained higher layer. Also, as init failure path is shared with exit path, both can get more testing.

Note though that when converting current calls or assignments to managed devm_* versions it is up to you to check if internal operations like allocating memory, have failed. Managed resources pertains to the freeing of these resources *only* - all other checks needed are still on you. In some cases this may mean introducing checks that were not necessary before moving to the managed devm_* calls.

3. Devres group

Devres entries can be grouped using devres group. When a group is released, all contained normal devres entries and properly nested groups are released. One usage is to rollback series of acquired resources on failure. For example:

```
   if (!devres_open_group(dev, NULL, GFP_KERNEL))
         return -ENOMEM;

   acquire A;
   if (failed)
         goto err;

   acquire B;
   if (failed)
         goto err;
   ...

   devres_remove_group(dev, NULL);
   return 0;

  err:
   devres_release_group(dev, NULL);
   return err_code;
```

As resource acquisition failure usually means probe failure, constructs like above are usually useful in midlayer driver (e.g. libata core layer) where interface function shouldn't have side effect on failure. For LLDs, just returning error code suffices in most cases.

Each group is identified by *void *id*. It can either be explicitly specified by @id argument to devres_open_group() or automatically created by passing NULL as @id as in the above example. In both cases, devres_open_group() returns the group's id. The returned id can be passed to other devres functions to select the target group. If NULL is given to those functions, the latest open group is selected.

For example, you can do something like the following:

```
  int my_midlayer_create_something()
  {
        if (!devres_open_group(dev, my_midlayer_create_something, GFP_KERNEL))
                return -ENOMEM;

        ...

        devres_close_group(dev, my_midlayer_create_something);
        return 0;
  }

  void my_midlayer_destroy_something()
```

```
{
	devres_release_group(dev, my_midlayer_create_something);
}
```

4. Details

Lifetime of a devres entry begins on devres allocation and finishes when it is released or destroyed (removed and freed) - no reference counting.

devres core guarantees atomicity to all basic devres operations and has support for single-instance devres types (atomic lookup-and-add-if-not-found). Other than that, synchronizing concurrent accesses to allocated devres data is caller's responsibility. This is usually non-issue because bus ops and resource allocations already do the job.

For an example of single-instance devres type, read pcim_iomap_table() in lib/devres.c.

All devres interface functions can be called without context if the right gfp mask is given.

5. Overhead

Each devres bookkeeping info is allocated together with requested data area. With debug option turned off, bookkeeping info occupies 16 bytes on 32bit machines and 24 bytes on 64bit (three pointers rounded up to ull alignment). If singly linked list is used, it can be reduced to two pointers (8 bytes on 32bit, 16 bytes on 64bit).

Each devres group occupies 8 pointers. It can be reduced to 6 if singly linked list is used.

Memory space overhead on ahci controller with two ports is between 300 and 400 bytes on 32bit machine after naive conversion (we can certainly invest a bit more effort into libata core layer).

6. List of managed interfaces

CLOCK
: devm_clk_get() devm_clk_get_optional() devm_clk_put() devm_clk_bulk_get() devm_clk_bulk_get_all() devm_clk_bulk_get_optional() devm_get_clk_from_childl() devm_clk_hw_register() devm_of_clk_add_hw_provider() devm_clk_hw_register_clkdev()

DMA
: dmaenginem_async_device_register() dmam_alloc_coherent() dmam_alloc_attrs() dmam_free_coherent() dmam_pool_create() dmam_pool_destroy()

DRM
: devm_drm_dev_alloc()

GPIO
: devm_gpiod_get() devm_gpiod_get_array() devm_gpiod_get_array_optional() devm_gpiod_get_index() devm_gpiod_get_index_optional() devm_gpiod_get_optional() devm_gpiod_put() devm_gpiod_unhinge() devm_gpiochip_add_data() devm_gpio_request() devm_gpio_request_one() devm_gpio_free()

I2C
: devm_i2c_new_dummy_device()

IIO
: devm_iio_device_alloc() devm_iio_device_register() devm_iio_dmaengine_buffer_setup() devm_iio_kfifo_buffer_setup() devm_iio_map_array_register() devm_iio_triggered_buffer_setup() devm_iio_trigger_alloc() devm_iio_trigger_register() devm_iio_channel_get() devm_iio_channel_get_all()

INPUT
: devm_input_allocate_device()

IO region
: devm_release_mem_region() devm_release_region() devm_release_resource() devm_request_mem_region() devm_request_region() devm_request_resource()

IOMAP
: devm_ioport_map() devm_ioport_unmap() devm_ioremap() devm_ioremap_uc() devm_ioremap_wc() devm_ioremap_np() devm_ioremap_resource() : checks resource, requests memory region, ioremaps devm_ioremap_resource_wc() devm_platform_ioremap_resource() : calls devm_ioremap_resource() for platform device devm_platform_ioremap_resource_byname() devm_platform_get_and_ioremap_resource() devm_iounmap() pcim_iomap() pcim_iomap_regions() : do request_region() and iomap() on multiple BARs pcim_iomap_table() : array of mapped addresses indexed by BAR pcim_iounmap()

IRQ
: devm_free_irq() devm_request_any_context_irq() devm_request_irq() devm_request_threaded_irq() devm_irq_alloc_descs() devm_irq_alloc_desc() devm_irq_alloc_desc_at() devm_irq_alloc_desc_from() devm_irq_alloc_descs_from() devm_irq_alloc_generic_chip() devm_irq_setup_generic_chip() devm_irq_sim_init()

LED
: devm_led_classdev_register() devm_led_classdev_unregister()

MDIO

devm_mdiobus_alloc() devm_mdiobus_alloc_size() devm_mdiobus_register() devm_of_mdiobus_register()

MEM

devm_free_pages() devm_get_free_pages() devm_kasprintf() devm_kcalloc() devm_kfree() devm_kmalloc() devm_kmalloc_array() devm_kmemdup() devm_krealloc() devm_kstrdup() devm_kvasprintf() devm_kzalloc()

MFD

devm_mfd_add_devices()

MUX

devm_mux_chip_alloc() devm_mux_chip_register() devm_mux_control_get() devm_mux_state_get()

NET

devm_alloc_etherdev() devm_alloc_etherdev_mqs() devm_register_netdev()

PER-CPU MEM

devm_alloc_percpu() devm_free_percpu()

PCI

devm_pci_alloc_host_bridge() : managed PCI host bridge allocation devm_pci_remap_cfgspace() : ioremap PCI configuration space devm_pci_remap_cfg_resource() : ioremap PCI configuration space resource pcim_enable_device() : after success, all PCI ops become managed pcim_pin_device() : keep PCI device enabled after release

PHY

devm_usb_get_phy() devm_usb_put_phy()

PINCTRL

devm_pinctrl_get() devm_pinctrl_put() devm_pinctrl_register() devm_pinctrl_unregister()

POWER

devm_reboot_mode_register() devm_reboot_mode_unregister()

PWM

devm_pwm_get() devm_of_pwm_get() devm_fwnode_pwm_get()

REGULATOR

devm_regulator_bulk_get() devm_regulator_get() devm_regulator_put() devm_regulator_register()

RESET

devm_reset_control_get() devm_reset_controller_register()

RTC

devm_rtc_device_register() devm_rtc_allocate_device() devm_rtc_register_device() devm_rtc_nvmem_register()

SERDEV

devm_serdev_device_open()

SLAVE DMA ENGINE

devm_acpi_dma_controller_register()

SPI

devm_spi_register_master()

WATCHDOG

devm_watchdog_register_device()