in linux/device.h

```
/* All 4 notifers below get called with the target struct device *
 * as an argument. Note that those functions are likely to be called
 * with the device lock held in the core, so be careful.
 */
#define BUS_NOTIFY_ADD_DEVICE       0x00000001 /* device added */
#define BUS_NOTIFY_DEL_DEVICE       0x00000002 /* device to be removed */
#define BUS_NOTIFY_REMOVED_DEVICE   0x00000003 /* device removed */
#define BUS_NOTIFY_BIND_DRIVER      0x00000004 /* driver about to be
                              bound */
#define BUS_NOTIFY_BOUND_DRIVER     0x00000005 /* driver bound to device */
#define BUS_NOTIFY_UNBIND_DRIVER    0x00000006 /* driver about to be
                              unbound */
#define BUS_NOTIFY_UNBOUND_DRIVER   0x00000007 /* driver is unbound
                              from the device */
```

当由device / driver被add到bus或从bus上remove时，kernel的driver framework会发出notification.

device相关的notification在drivers/base/core.c中处理。

```
#define BUS_NOTIFY_ADD_DEVICE       0x00000001 /* device added */
#define BUS_NOTIFY_DEL_DEVICE       0x00000002 /* device to be removed */
#define BUS_NOTIFY_REMOVED_DEVICE   0x00000003 /* device removed */
```

`BUS_NOTIFY_ADD_DEVICE`

在device被添加到bus后但在binding driver以前

```c
int device_add(struct device *dev)
{
    struct device *parent = NULL;
    struct kobject *kobj;
    struct class_interface *class_intf;
    int error = -EINVAL;

    dev = get_device(dev);
    if (!dev)
        goto done;

    if (!dev->p) {
        error = device_private_init(dev);
        if (error)
            goto done;
    }

    /*
     * for statically allocated devices, which should all be converted
     * some day, we need to initialize the name. We prevent reading back
     * the name, and force the use of dev_name()
     */
    if (dev->init_name) {
        dev_set_name(dev, "%s", dev->init_name);
        dev->init_name = NULL;
    }

    /* subsystems can specify simple device enumeration */
    if (!dev_name(dev) && dev->bus && dev->bus->dev_name)
        dev_set_name(dev, "%s%u", dev->bus->dev_name, dev->id);

    if (!dev_name(dev)) {
        error = -EINVAL;
        goto name_error;
    }

    pr_debug("device: '%s': %s\n", dev_name(dev), __func__);

    parent = get_device(dev->parent);
    kobj = get_device_parent(dev, parent);
    if (kobj)
        dev->kobj.parent = kobj;

    /* use parent numa_node */
    if (parent)
        set_dev_node(dev, dev_to_node(parent));

    /* first, register with generic layer. */
    /* we require the name to be set before, and pass NULL */
    error = kobject_add(&dev->kobj, dev->kobj.parent, NULL);
    if (error)
        goto Error;

    /* notify platform of device entry */
    if (platform_notify)
        platform_notify(dev);

    error = device_create_file(dev, &dev_attr_uevent);
    if (error)
        goto attrError;

    if (MAJOR(dev->devt)) {
        error = device_create_file(dev, &dev_attr_dev);
        if (error)
            goto ueventattrError;

        error = device_create_sys_dev_entry(dev);
        if (error)
            goto devtattrError;

        devtmpfs_create_node(dev);
    }

    error = device_add_class_symlinks(dev);
    if (error)
        goto SymlinkError;
    error = device_add_attrs(dev);
    if (error)
        goto AttrsError;
    error = bus_add_device(dev);
    if (error)
        goto BusError;
    error = dpm_sysfs_add(dev);
    if (error)
        goto DPMError;
    device_pm_add(dev);

    /* Notify clients of device addition.  This call must come
     * after dpm_sysfs_add() and before kobject_uevent().
     */
    if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,    ①
                     BUS_NOTIFY_ADD_DEVICE, dev);

    kobject_uevent(&dev->kobj, KOBJ_ADD);
    bus_probe_device(dev);                                          ②
    if (parent)
        klist_add_tail(&dev->p->knode_parent,
                   &parent->p->klist_children);

    if (dev->class) {
        mutex_lock(&dev->class->p->mutex);
        /* tie the class to the device */
        klist_add_tail(&dev->knode_class,
                   &dev->class->p->klist_devices);

        /* notify any interfaces that the device is here */
        list_for_each_entry(class_intf,
                    &dev->class->p->interfaces, node)
            if (class_intf->add_dev)
                class_intf->add_dev(dev, class_intf);
        mutex_unlock(&dev->class->p->mutex);
    }

    ......
}
```

在发出 `BUS_NOTIFY_ADD_DEVICE` 以前，该device相关在sysfs中的文件都已经建立完毕了，但还没有bingding driver。

①
发出BUS_NOTIFY_ADD_DEVICE notification

②
这才是match device and driver，并由driver probe device的函数。

即BUS_NOTIFY_ADD_DEVICE notification在driver的probe()运行以前发出。

`BUS_NOTIFY_DEL_DEVICE`

`BUS_NOTIFY_REMOVED_DEVICE`

在真正delete device以前，即device还处于完好状态时发出 `BUS_NOTIFY_DEL_DEVICE` notification.
而在清理动作完成后发出 `BUS_NOTIFY_REMOVED_DEVICE` notification.

```c
void device_del(struct device *dev)
{
    struct device *parent = dev->parent;
    struct class_interface *class_intf;

    /* Notify clients of device removal.  This call must come
     * before dpm_sysfs_remove().
     */
    if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                                     BUS_NOTIFY_DEL_DEVICE, dev);
    dpm_sysfs_remove(dev);
    if (parent)
        klist_del(&dev->p->knode_parent);
    if (MAJOR(dev->devt)) {
        devtmpfs_delete_node(dev);
        device_remove_sys_dev_entry(dev);
        device_remove_file(dev, &dev_attr_dev);
    }
    if (dev->class) {
        device_remove_class_symlinks(dev);

        mutex_lock(&dev->class->p->mutex);
        /* notify any interfaces that the device is now gone */
        list_for_each_entry(class_intf,
                    &dev->class->p->interfaces, node)
            if (class_intf->remove_dev)
                class_intf->remove_dev(dev, class_intf);
        /* remove the device from the class list */
        klist_del(&dev->knode_class);
        mutex_unlock(&dev->class->p->mutex);
    }
    device_remove_file(dev, &dev_attr_uevent);
    device_remove_attrs(dev);
    bus_remove_device(dev);
    device_pm_remove(dev);
    driver_deferred_probe_del(dev);

    /* Notify the platform of the removal, in case they
     * need to do anything...
     */
    if (platform_notify_remove)
        platform_notify_remove(dev);
    if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                                     BUS_NOTIFY_REMOVED_DEVICE, dev);
    kobject_uevent(&dev->kobj, KOBJ_REMOVE);
    cleanup_device_parent(dev);
    kobject_del(&dev->kobj);
    put_device(parent);
}
```

in drivers/base/dd.c

```c
/**
 * device_attach - try to attach device to a driver.
 * @dev: device.
 *
 * Walk the list of drivers that the bus has and call
 * driver_probe_device() for each pair. If a compatible
 * pair is found, break out and return.
 *
 * Returns 1 if the device was bound to a driver;
 * 0 if no matching driver was found;
 * -ENODEV if the device is not registered.
 *
 * When called for a USB interface, @dev->parent lock must be held.
 */
int device_attach(struct device *dev)
{
    int ret = 0;

    device_lock(dev);
    if (dev->driver) {
        if (klist_node_attached(&dev->p->knode_driver)) {
            ret = 1;
            goto out_unlock;
        }
        ret = device_bind_driver(dev);
        if (ret == 0)
            ret = 1;
        else {
            dev->driver = NULL;
            ret = 0;
        }
    } else {
        ret = bus_for_each_drv(dev->bus, NULL, dev, __device_attach);
        pm_request_idle(dev);
    }
out_unlock:
    device_unlock(dev);
    return ret;
}
```

device 与　driver之间binding的核心函数是 `device_bind_driver()`

```
int device_bind_driver(struct device *dev)
{
    int ret;

    ret = driver_sysfs_add(dev);
    if (!ret)
        driver_bound(dev);
    return ret;
}
```

在driver_sysfs_add()中发出 `BUS_NOTIFY_BIND_DRIVER` notification.

在driver_bound()的最后 `BUS_NOTIFY_BOUND_DRIVER` notification.

```
static void driver_bound(struct device *dev)
{
    if (klist_node_attached(&dev->p->knode_driver)) {
        printk(KERN_WARNING "%s: device %s already bound\n",
            __func__, kobject_name(&dev->kobj));
        return;
    }

    pr_debug("driver: '%s': %s: bound to device '%s'\n", dev->driver->name,
         __func__, dev_name(dev));

    klist_add_tail(&dev->p->knode_driver, &dev->driver->p->klist_devices);

    /*
     * Make sure the device is no longer in one of the deferred lists and
     * kick off retrying all pending devices
     */
    driver_deferred_probe_del(dev);
    driver_deferred_probe_trigger();        ①

    if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,    ②
                         BUS_NOTIFY_BOUND_DRIVER, dev);
}
```

①
会触发deferred_probe_work_func()运行。

```
/*
 * deferred_probe_work_func() - Retry probing devices in the active list.
 */
static void deferred_probe_work_func(struct work_struct *work)
{
    struct device *dev;
    struct device_private *private;
    /*
     * This block processes every device in the deferred 'active' list.
     * Each device is removed from the active list and passed to
     * bus_probe_device() to re-attempt the probe.  The loop continues
     * until every device in the active list is removed and retried.
     *
     * Note: Once the device is removed from the list and the mutex is
     * released, it is possible for the device get freed by another thread
     * and cause a illegal pointer dereference.  This code uses
     * get/put_device() to ensure the device structure cannot disappear
     * from under our feet.
     */
    mutex_lock(&deferred_probe_mutex);
    while (!list_empty(&deferred_probe_active_list)) {
        private = list_first_entry(&deferred_probe_active_list,
                    typeof(*dev->p), deferred_probe);
        dev = private->device;
        list_del_init(&private->deferred_probe);

        get_device(dev);

        /*
         * Drop the mutex while probing each device; the probe path may
         * manipulate the deferred list
         */
        mutex_unlock(&deferred_probe_mutex);

        /*
         * Force the device to the end of the dpm_list since
         * the PM code assumes that the order we add things to
         * the list is a good order for suspend but deferred
         * probe makes that very unsafe.
         */
        device_pm_lock();
        device_pm_move_last(dev);
        device_pm_unlock();

        dev_dbg(dev, "Retrying from deferred list\n");
        bus_probe_device(dev);

        mutex_lock(&deferred_probe_mutex);

        put_device(dev);
    }
    mutex_unlock(&deferred_probe_mutex);
}
```

这里的核心是 `bus_probe_device(dev);` 即driver match device并probe device。

②
发出BUS_NOTIFY_BOUND_DRIVER notification。感觉BUS_NOTIFY_BOUND_DRIVER的发出与该device是否
已经与driver bound是异步的。