

Kernel Course: Lecture 16

The Linux Device Model

Illia Smyrnov

May 24, 2020

GlobalLogic

Agenda

1. The Linux Kernel Device Model
2. Assignments

The Linux Kernel Device Model

Brief history

- The demands of newer systems, with their more complicated topologies and need to support features such as power management, made it clear that a general abstraction describing the structure of the system was needed.
- A new device driver model has been designed and developed by Patrick Mochel in 2.5 Linux kernel development series.
- The model has been included in the mainstream kernel since version 2.5.1.
- The driver model was created by analyzing the behavior of the PCI and USB bus drivers.
- The original intent of the model was to provide a means to generically represent and operate on every device in a system.
- During the integration of the initial model, it became apparent that many more concepts related to devices could be generalized and shared.

Power management and system shutdown

These require an understanding of the system's structure. The device model enables a traversal of the system's hardware in the right order.

Communications with user space

The implementation of the sysfs virtual filesystem is tightly tied into the device model and exposes the structure represented by it.

Hotpluggable devices

Peripherals can come and go at the whim of the user. The hotplug mechanism used within the kernel to handle and (especially) communicate with user space about the plugging and unplugging of devices is managed through the device model.

Use cases (cont'd)

Device classes

Many parts of the system have to know what kinds of devices are available. The device model includes a mechanism for assigning devices to classes, which describe devices at a higher, functional level.

Object lifecycles

Many of the functions (like hotplug support) require creation and manipulation of objects within the kernel. The implementation of the device model offers set of mechanisms for dealing with object lifecycles.

Example: USB mouse connected to BBB

The Linux device model is a complex data structure. For example, the figure below shows in simplified form a tiny piece of the device model structure associated with a USB mouse connected to BeagleBone Black.

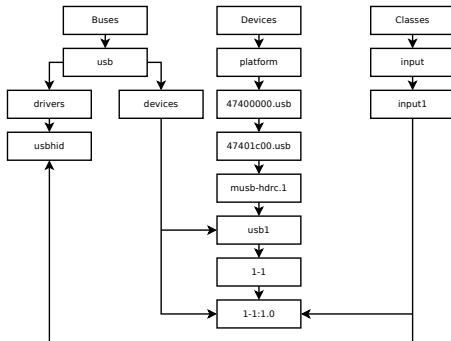


Figure 1: A tiny piece of the device model structure

Kobject in the device model

The kobject is the fundamental structure that holds the device model together. Major tasks handled by **struct** kobject and its supporting code:

Reference counting of objects

Often, when a kernel object is created, there is no way to know just how long it will exist. One way of tracking the lifecycle of such objects is through reference counting. When no code in the kernel holds a reference to a given object, that object has finished its useful life and can be deleted.

Sysfs representation

Every object that shows up in sysfs has a kobject that interacts with the kernel to create its visible representation.

Kobject in the device model (cont'd)

Data structure glue

The device model is, a complicated data structure made up of multiple hierarchies with numerous links between them. The kobject implements this structure and holds it together.

Hotplug event handling

The kobject subsystem handles the generation of events that notify user space about the comings and goings of hardware on the system.

Kobject structure

is defined in `<linux/kobject.h>` with number of other related structures and functions for manipulating them.

```
struct kobject {  
    const char          *name;  
    struct list_head    entry;  
    struct kobject      *parent;  
    struct kset          *kset;  
    struct kobj_type     *ktype;  
    struct kernfs_node   *sd; /* sysfs directory entry */  
    struct kref          kref;  
    /* ... */  
};
```

Object is designed to be embedded

into higher-level objects. Kobjects are generally not interesting on their own.. Instead, they are usually embedded within some other structure which contains the stuff is really interested in. For an example below is `struct cdev`

```
struct cdev {  
    struct kobject kobj;  
    struct module *owner;  
    const struct file_operations *ops;  
    struct list_head list;  
    dev_t dev;  
    unsigned int count;  
};
```

Only one kobject could be embedded

No structure should EVER have more than one kobject embedded within it. If it does, the reference counting for the object is sure to be messed up and incorrect [1].

To find the containing structure

code that works with kobjects should use `container_of` macro, as shown in the following example:

```
/*...*/  
struct cdev *p = container_of(kobj, struct cdev, kobj);  
/*...*/
```

Kobject - operations

- `void kobject_init(struct kobject *kobj);`
 - Kobject **must be zeroed out** before passed to this function
 - Kobject parent and kset should be set before a call to `kobject_init()`
- `struct kobject *kobject_create(void);`
 - Dynamically create kobject and zeroing it
 - Call `kobject_init()`
- `struct kobject *kobject_create_and_add(const char *name, struct kobject *parent);`
 - Do the same as `kobject_create()`
 - Automatically registers kobject with sysfs

Kobject - operations (con'd)

- `int kobject_rename(struct kobject *kobj, const char *new_name);`
 - As the name of the kobject (`kobj->name`) is set when it is added to the kernel, the name of the kobject should never be manipulated directly
 - `kobject_rename()` does not perform any locking or have a solid notion of what names are valid so the caller must provide their own sanity checking
- `struct kobject *kobject_get(struct kobject *kobj);`
- `void kobject_put(struct kobject *kobj);`
 - Get / put references to a kobject
 - Mostly wrappers around `kref_get()` / `kref_put()`
 - Calls kobject's `ktype release()` method if refcount reaches 0
- `static void kobject_release(struct kref *kref);`
 - Used to destroy a kobject, called from `kobject_put()`.

Ktypes and release methods

`kobj_type` (`ktype`)

is used to describe a particular type of kobject (or, more correctly, of containing object). Every kobject needs to have an associated `kobj_type` structure (defined in `<linux/kobject.h>`)

```
struct kobj_type {  
    void (*release)(struct kobject *kobj);  
    const struct sysfs_ops *sysfs_ops;  
    struct attribute **default_attrs;  
    const struct kobj_ns_type_operations  
        *(*child_ns_type)(struct kobject *kobj);  
    const void *(*namespace)(struct kobject *kobj);  
};
```

Ktypes and release methods (con'd)

kobj_type defines a default behavior

for objects which embed kobject through a list of default attributes. Different kobjects with the same ktype are enabled to share the same behavior

kobj_type stores **release()** method

used to free structure protected by a kobject when its reference count goes to zero.

The reference count is not under the direct control of the code which created the kobject. So that code must be notified asynchronously whenever the last reference to one of its kobjects goes away. This notification is done through a kobject's **release()** method. Every kobject must have a **release()** method, and the kobject must persist (in a consistent state) until that method is called.

Ktypes and release methods (con'd)

Usually `release()` method has a form like this:

```
void my_object_release(struct kobject *kobj)
{
    struct my_object *mine = container_of(kobj, struct my_object,
                                           kobj);

    /* Perform any additional cleanup on this object */
    /* ... */
    kfree(mine);
}
```

kset

is a collection of kobjects embedded within structures of the same type. These kobjects can be of the same ktype or belong to different ktypes. kset structure (defined in <linux/kobject.h>)

```
struct kset {  
    struct list_head list;  
    spinlock_t list_lock;  
    struct kobject kobj;  
    const struct kset_uevent_ops *uevent_ops;  
};
```

Following fields and methods are defined in `kset` structure:

- `list` is a linked list of all `kobjects` in this `kset`;
- `list_lock` is a spinlock protecting this entry in the list;
- `kobj` is a `kobject` representing the base class for this set;
- `uevent_ops` points to a structure that describes the hotplug behavior of `kobjects` in this `kset`.

A kset serves the following functions:

- act as a bag containing a group of objects. A kset can be used by the kernel to track "all block devices" or "all PCI device drivers"
- A kset is also a subdirectory in sysfs, where the associated kobjects with the kset can show up. Every kset contains a kobject which can be set up to be the parent of other kobjects. The top-level directories of the sysfs hierarchy are constructed in this way
- Ksets can support the "hotplugging" of kobjects and influence how uevent events are reported to user space.

Ksets (con'd)

A kset keeps its children in a standard kernel linked list. In almost all cases, the contained kobjects also have pointers to the kset (or, strictly, its embedded kobject) in their parent's fields. Typically, a kset and its kobjects look something like in the figure below:

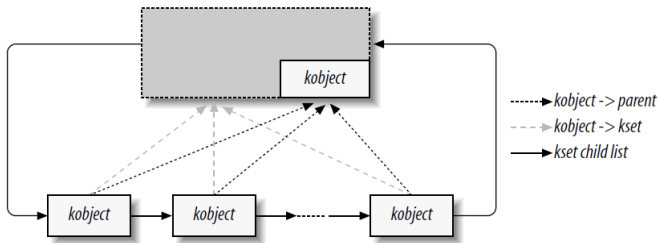


Figure 2: Simple kset hierarchy.

Kset - operations

For initialization and setup, ksets have an interface very similar to that of kobjects:

- `static struct kset *kset_create(const char *name, const struct kset_uevent_ops *uevent_ops, struct kobject *parent_kobj);`
 - dynamically creates a kset;
 - prepare kset to be used by `kset_register();`
- `void kset_init(struct kset *k);`
 - initialize the kset for usage (setup kset's list, initialize the spinlock, initialize the internal kset's kobject);
 - you will almost never need to use it;
- `int kset_register(struct kset *k);`
 - call `kset_init();`
 - register kset's internal kobject;

Kset - operations (con'd)

- `struct kset *kset_create_and_add(const char *name, const struct kset_uevent_ops *uevent_ops, struct kobject *parent_kobj);`
 - creates a kset structure dynamically, call `kset_create()`;
 - registers it with sysfs (call `kset_register()`)
- `void kset_unregister(struct kset *k);`
 - remove a kset from sysfs;
 - decrements reference count (of internal kobject).

Buses

A bus is a channel between the processor and one or more devices. For the purposes of the device model, all devices are connected via a bus, even if it is an internal, virtual, "platform" bus. Buses can plug into each other. A USB controller is often a PCI device, for example. The device model represents the actual connections between buses and the devices they control. A bus is represented by the `bus_type` structure, defined in `<linux/device.h>`. The major tasks handled by a bus:

- binds devices and drivers;
- matching;
- uevents;
- handle shutdown.

Buses (con'd)

bus_type structure contains the name, the default attributes, the bus' methods, PM operations, and the driver core's private data.

```
struct bus_type {
    const char          *name;
    const char          *dev_name;
    struct device        *dev_root;
    const struct attribute_group **bus_groups;
    const struct attribute_group **dev_groups;
    const struct attribute_group **drv_groups;

    int (*match)(struct device *dev, struct device_driver *drv);
    /*...*/
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    /*...*/
};
```

Devices

At the lowest level, every device in a Linux system is represented by an instance of `struct device`. The device structure contains the information that the device model core needs to model the system. Most subsystems, track additional information about the devices they host. As a result, device structure is usually embedded within a higher-level representation of the device. For example, definitions of `struct usb_device`, contains `struct device` buried inside:

```
struct usb_device {  
    int      devnum;  
    char      devpath[16];  
    /*...*/  
    struct device dev;  
    /*...*/  
};
```

Devices (con'd)

device structure contains the name, the default attributes, the bus' methods, PM operations, and the driver core's private data.

```
struct device {
    struct device      *parent;
    struct device_private *p;
    struct kobject kobj;
    const char      *init_name;
    const struct device_type *type;
    /*...*/
    struct bus_type      *bus;
    struct device_driver *driver;
    /*...*/
    struct class      *class;
    const struct attribute_group **groups;
    /*...*/
    void (*release)(struct device *dev);
}
```

Device Drivers

The device model tracks all of the drivers known to the system. The main reason for this tracking is to enable the driver core to match up drivers with new devices. Once drivers are known objects within the system, a number of other things become possible.

Device drivers can export information and configuration variables that are independent of any specific device. As in the case with most driver core structures, the `device_driver` structure is usually found embedded within a higher-level, bus-specific structure. The major tasks handled by a device driver:

- controls a device;
- probe/remove
- PM (suspend/resume, shutdown)
- default attributes

Device Drivers (con'd)

device_driver structure contains the name, the default attributes, the driver's methods, PM operations, driver core's private data.

```
struct device_driver {
    const char                *name;
    struct bus_type           *bus;
    struct module             *owner;
    const char                *mod_name;
/*...*/
    const struct of_device_id *of_match_table;
/*...*/
    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
/*...*/
    const struct attribute_group **groups;
/*...*/
};
```

Classes

A class is a higher-level view of a device that abstracts out low-level implementation details. Drivers may see a SCSI disk or an ATA disk, but, at the class level, they are all simply disks. Classes allow user space to work with devices based on what they do, rather than how they are connected or how they work.

- Almost all classes show up in sysfs under `/sys/class`. Thus, for example, all network interfaces can be found under `/sys/class/net`, regardless of the type of interface. Input devices can be found in `/sys/class/input`, and serial devices are in `/sys/class/tty`.
- The one **exception is block devices**, which can be found under `/sys/block` for historical reasons.
- Class membership is usually handled by high-level code without the need for explicit support from drivers.

Classes (con'd)

A class is defined by an instance of **struct** class:

```
struct class {  
    const char      *name;  
    struct module   *owner;  
    const struct attribute_group  **class_groups;  
    const struct attribute_group  **dev_groups;  
    struct kobject   *dev_kobj;  
  
    int (*dev_uevent)(struct device *dev, struct kobj_uevent_env *env);  
/*...*/  
    char *(*devnode)(struct device *dev, umode_t *mode);  
    void (*class_release)(struct class *class);  
/*...*/  
};
```

Sysfs filesystem

- The sysfs filesystem is an in-memory virtual filesystem
- Exports to userspace a friendly view of the kobjects hierarchy
- It enables users to view the device topology of their system as a simple filesystem
- Each sysfs directory corresponds to a single kobject
- Each sysfs file corresponds to a kobject's attribute
- Using attributes, kobjects can export files that enable kernel variables to be read from and optionally written to.

Sysfs entries for kobjects are always directories

so a call to `kobject_add()` results in the creation of a directory in sysfs. Usually that directory contains one or more attributes (will be described later).

The name assigned to the kobject is used for sysfs directory.

Kobjects that appear in the same sysfs hierarchy must have unique names. Names assigned to kobjects should be reasonable file names: they cannot contain the slash character, and the use of white space is strongly discouraged.

The sysfs entry is located in the directory corresponding to the kobject's parent

pointer. If parent is NULL when `kobject_add()` is called, it is set to the kobject embedded in the new kobject's kset; thus, the sysfs hierarchy usually matches the internal hierarchy created with ksets. If both parent and kset are NULL, the sysfs directory is created at the top level.

Low-Level Sysfs Operations - Default Attributes

When created, every kobject is given a set of default attributes. These attributes are specified by the `kobj_type` structure. The `default_attrs` field lists the attributes to be created for every kobject of this type, and `sysfs_ops` provides the methods to implement those attributes.

```
struct kobj_type {  
    void (*release)(struct kobject *kobj);  
    const struct sysfs_ops *sysfs_ops;  
    struct attribute **default_attrs;  
    const struct kobj_ns_type_operations  
        *(*child_ns_type)(struct kobject *kobj);  
    const void *(*namespace)(struct kobject *kobj);  
};
```

Low-Level Sysfs Operations - Default Attributes (con'd)

The `default_attrs` field of `kobj_type` structure points to an array of pointers to attribute structures.

- The `name` is the name of the attribute (as it appears within the kobject's sysfs directory)
- The `mode` is the protection bits that are to be applied to this attribute (usually `S_IRUGO` for read-only attributes and `S_IWUSR` to give write access to root only). The last entry in the `default_attrs` list **must be zero-filled**.

```
struct attribute {  
    const char    *name;  
    umode_t       mode;  
    /* ... */  
};
```

Low-Level Sysfs Operations - Default Attributes (con'd)

The `sysfs_ops` field of `kobj_type` structure points `sysfs_ops` structure.

```
struct sysfs_ops {  
    ssize_t (*show)(struct kobject *kobj, struct attribute *attr,  
                    char *buffer);  
    ssize_t (*store)(struct kobject *kobj, struct attribute *attr,  
                    const char *buffer, size_t size);  
};
```

There are two methods:

- The `show()` method is invoked when the sysfs entry is read from user-space
- The `store()` method is invoked on write.

Low-Level Sysfs Operations - Default Attributes (con'd)

The `show()` and `store()` methods should be implemented as following:

- The `show()` must copy the value of the attribute given by `attr` into the buffer provided by `buffer`. The `buffer` is `PAGE_SIZE` bytes in length. The function should return the size in bytes of data actually written into buffer on success or a negative error code on failure
- The `store()` method is invoked on write. It must read the `size` bytes from `buffer` into the variable represented by the attribute `attr`. The size of the `buffer` is always `PAGE_SIZE` or smaller. The function should return the size in bytes of data read from `buffer` on success or a negative error code on failure.

Low-Level Sysfs Operations - Creating New Attributes

The kernel provides the `sysfs_create_file()` function for adding new attributes on top of the default set.

```
int sysfs_create_file(struct kobject *kobj,  
                     const struct attribute *attr);
```

In addition to creating actual files, `sysfs_create_link()` is provided to create symbolic links.

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target,  
                     const char *name);
```

Low-Level Sysfs Operations - Binary Attributes

In some cases data must be passed, untouched, between user space and the device (e.g. uploading firmware, etc). Binary attributes are supported to handle such tasks. They are described with a `bin_attribute` structure:

```
struct bin_attribute {
    struct attribute      attr;
    size_t               size;
    void                 *private;
    ssize_t (*read)(struct file *, struct kobject *,
                    struct bin_attribute *, char *, loff_t, size_t);
    ssize_t (*write)(struct file *, struct kobject *,
                     struct bin_attribute *, char *, loff_t, size_t);
    int (*mmap)(struct file *, struct kobject *,
                struct bin_attribute *attr, struct vm_area_struct *vma);
};
```

Low-Level Sysfs Operations - Binary Attributes (con'd)

Following fields and methods are defined in `bin_attribute` structure:

- `attr` is an attribute structure giving the name, owner and permissions for the binary attribute
- `size` is the maximum size of the binary attribute (or 0 if there is no maximum).
- `read()`, `write()` and `mmap()` methods work similarly to the normal char driver equivalents.

Note: `read()`, `write()` can be called multiple times for a single load with a maximum of one page worth of data in each call.

Low-Level Sysfs Operations - Binary Attributes (con'd)

Binary attributes must be created explicitly. They cannot be set up as default attributes. To create a binary attribute, call:

```
int sysfs_create_bin_file(struct kobject *kobj,  
                          const struct bin_attribute *attr)
```

Binary attributes can be removed with:

```
void sysfs_remove_bin_file(struct kobject *kobj,  
                           const struct bin_attribute *attr)
```

Low-Level Sysfs Operations - Attribute groups

It is often necessary to add multiple sysfs files to be created and destroyed all at once. Attribute groups is the solution. Attribute group could be described by instance of `struct attribute_group`. It contain optional `name` field, arrays of attributes and binary attributes and corresponding optional functions to control the attribute's mode and visibility.

```
struct attribute_group {
    const char    *name;
    umode_t      (*is_visible)(struct kobject *,
                               struct attribute *, int);
    umode_t      (*is_bin_visible)(struct kobject *,
                                   struct bin_attribute *, int);
    struct attribute **attrs;
    struct bin_attribute **bin_attrs;
};
```

Example: USB flash drive connected to BBB

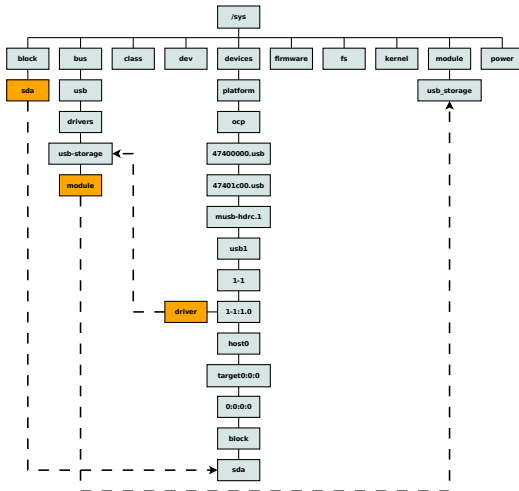


Figure 3: sysfs tree related to USB flash drive connected to BBB

sysfs Conventions

sysfs attributes should export one value per file.

Values should be text-based and map to simple C types. If one-value-per-file rule results in an inefficient data representation, it is acceptable to place multiple values of the same type in one file. Delineate them as appropriate (a simple space probably makes the most sense)

Organize data in sysfs in a clean hierarchy.

Correctly parent kobjects, associate attributes with the correct kobject. Keep the sysfs tree organized and hierarchical.

sysfs provides a kernel-to-user service and is thus a sort of user-space ABI.

User programs can rely on the existence, location, value, and behavior of sysfs directories and files. Changing existing files in any way is discouraged, and modifying the behavior of a given attribute but keeping its name and location is surely begging for trouble.

The Kernel Events Layer

The Kernel Event Layer implements a kernel-to-user notification system on top of kobjects.

- It models events as **signals** emitting from objects - specifically, kobjects.
- Because kobjects map to sysfs paths, the **source** of each event is a sysfs path.
- Each event is given a **verb or action** string representing the **signal**. The strings are terms such as modified or unmounted that describe what happened.
- Each event has an **optional payload**. Rather than pass an arbitrary string to user-space that provides the payload, the kernel event layer represents payloads as sysfs attributes.

The Kernel Events Layer (cont'd)

All the supported verbs are defined in `kobject_action` enumeration:

```
enum kobject_action {  
    KOBJ_ADD,  
    KOBJ_REMOVE,  
    KOBJ_CHANGE,  
    KOBJ_MOVE,  
    KOBJ_ONLINE,  
    KOBJ_OFFLINE,  
    KOBJ_BIND,  
    KOBJ_UNBIND,  
    KOBJ_MAX  
};
```

The Kernel Events Layer (cont'd)

There are two ways to pass kernel events from kernel-space out to user-space:

via netlink

Netlink is a high-speed multicast socket that transmits networking information.; Using netlink means that obtaining kernel events from user-space is as simple as blocking on a socket

by uevent helper program

The uevent helper program is forked by the kernel for every uevent. Before the switch to the netlink-based uevent source, this was used to hook hotplug scripts into kernel device events. It usually pointed to a shell script at /sbin/hotplug. This should not be used today, because usual systems create many events at bootup or device discovery in a very short time frame. One forked process per event can create so many processes that it creates a high system load, or on smaller systems it is known to create out-of-memory situations during bootup.

The Kernel Events Layer (cont'd)

The helper program mechanism is still used for small systems. Following commands provided in `/etc/init.d/rcS` to:

- mount `sysfs` to `/sys`;
- setup `uevent` helper program (`mdev`);
- process all the devices appeared before the helper program was set (`mdev -s`).

```
#!/bin/sh
```

```
mount -t sysfs none /sys
```

```
# ...
```

```
echo /sbin/mdev > /proc/sys/kernel/hotplug
```

```
mdev -s
```

The Kernel Events Layer (cont'd)

The code fragments (from `uevent-dump.c` sample) shows how obtain kernel events using netlink.

```
/*...*/
fd = socket(PF_NETLINK, SOCK_RAW, NETLINK_KOBJECT_UEVENT);
/*...*/
nls.nl_family = AF_NETLINK;
nls.nl_pid = getpid();
nls.nl_groups = 1;
res = bind(fd, (struct sockaddr *)&nls, sizeof(nls));
/*...*/
while (1) {
    len = recv(fd, buf, sizeof(buf), 0);
    /*...*/
}
/*...*/
```

The Kernel Events Layer (cont'd)

On PC you can monitor uevents using `udevadm` as below, dump fragment shows one uevent from kernel after new USB device has been connected:

```
udevadm monitor --env
```

monitor will print the received events **for**:

UDEV - the event which udev sends out after rule processing

KERNEL - the kernel uevent

```
KERNEL[42591.507577] add      /devices/pci0000:00/0000:00:1c.6/0000:0e:00.0/usb1/1-2/1-2.1 (usb)
ACTION=add
BUSNUM=001
DEVNAME=/dev/bus/usb/001/014
DEVNUM=014
DEVPATH=/devices/pci0000:00/0000:00:1c.6/0000:0e:00.0/usb1/1-2/1-2.1
DEVTYPE=usb_device
MAJOR=189
MINOR=13
PRODUCT=45e/40/300
SEQNUM=2485
SUBSYSTEM=usb
TYPE=0/0/0
```

Example: kobject and attributes

Lets learn an example how to use kobject directly to create attributes in sysfs. The sample is provided by Greg Kroah-Hartman and included in to Linux source tree, file:samples/kobject/kobject-example.c.

Please note - the example below is only for demonstration. If you are driver writer:

- use attribute groups only;
- never call `sysfs_*()`;
- never create sysfs files individually.

Example: using kobject directly (cont'd)

```
// SPDX-License-Identifier: GPL-2.0
/*
 * Sample kobject implementation
 *
 * Copyright (C) 2004-2007 Greg Kroah-Hartman <greg@kroah.com>
 * Copyright (C) 2007 Novell Inc.
 */
#include <linux/kobject.h>
#include <linux/string.h>
#include <linux/sysfs.h>
#include <linux/module.h>
#include <linux/init.h>

/*
 * This module shows how to create a simple subdirectory in sysfs called
 * /sys/kernel/kobject-example. In that directory, 3 files are created:
 * "foo", "baz", and "bar". If an integer is written to these files, it can be
 * later read out of it.
 */

static int foo;
static int baz;
static int bar;
```

Example: using kobject directly (cont'd)

```
/*
 * The "foo" file where a static variable is read from and written to.
 */
static ssize_t foo_show(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf)
{
    return sprintf(buf, "%d\n", foo);
}

static ssize_t foo_store(struct kobject *kobj, struct kobj_attribute *attr,
                        const char *buf, size_t count)
{
    int ret;

    ret = kstrtoint(buf, 10, &foo);
    if (ret < 0)
        return ret;

    return count;
}

/* Sysfs attributes cannot be world-writable. */
static struct kobj_attribute foo_attribute =
    __ATTR(foo, 0664, foo_show, foo_store);
```

Example: using kobject directly (cont'd)

```
/*
 * More complex function where we determine which variable is being accessed by
 * looking at the attribute for the "baz" and "bar" files.
 */
static ssize_t b_show(struct kobject *kobj, struct kobj_attribute *attr,
                      char *buf)
{
    int var;

    if (strcmp(attr->attr.name, "baz") == 0)
        var = baz;
    else
        var = bar;
    return sprintf(buf, "%d\n", var);
}
```

Example: using kobject directly (cont'd)

```
static ssize_t b_store(struct kobject *kobj, struct kobj_attribute *attr,
                      const char *buf, size_t count)
{
    int var, ret;

    ret = kstrtoint(buf, 10, &var);
    if (ret < 0)
        return ret;

    if (strcmp(attr->attr.name, "baz") == 0)
        baz = var;
    else
        bar = var;
    return count;
}

static struct kobj_attribute baz_attribute =
    __ATTR(baz, 0664, b_show, b_store);
static struct kobj_attribute bar_attribute =
    __ATTR(bar, 0664, b_show, b_store);
```

Example: using kobject directly (cont'd)

```
/*
 * Create a group of attributes so that we can create and destroy them all
 * at once.
 */
static struct attribute *attrs[] = {
    &foo_attribute.attr,
    &baz_attribute.attr,
    &bar_attribute.attr,
    NULL, /* need to NULL terminate the list of attributes */
};

/*
 * An unnamed attribute group will put all of the attributes directly in
 * the kobject directory.  If we specify a name, a subdirectory will be
 * created for the attributes with the directory being the name of the
 * attribute group.
 */
static struct attribute_group attr_group = {
    .attrs = attrs,
};
```

Example: using kobject directly (cont'd)

```
static struct kobject *example_kobj;

static int __init example_init(void)
{
    int retval;

    /*
     * Create a simple kobject with the name of "kobject_example",
     * located under /sys/kernel/
     *
     * As this is a simple directory, no uevent will be sent to
     * userspace. That is why this function should not be used for
     * any type of dynamic kobjects, where the name and number are
     * not known ahead of time.
     */
    example_kobj = kobject_create_and_add("kobject_example", kernel_kobj);
    if (!example_kobj)
        return -ENOMEM;

    /* Create the files associated with this kobject */
    retval = sysfs_create_group(example_kobj, &attr_group);
    if (retval)
        kobject_put(example_kobj);

    return retval;
}
```

Example: using kobject directly (cont'd)

```
static void __exit example_exit(void)
{
    kobject_put(example_kobj);
}

module_init(example_init);
module_exit(example_exit);
MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Greg Kroah-Hartman <greg@kroah.com>");
```

How to Create a sysfs Files Correctly

Consider the following fragment of the driver code:

```
/* ... */
int my_driver_probe(...)
{
    /*...*/
    retval = sysfs_create_group(&my_device->kobj, &my_attribute_group);
    if (retval)
        goto error;
    /*...*/
    return 0;

error:
    /* Clean up other things and return an error */
    /*...*/
    return -ENODEV;
}
/* ... */
```

Things seem to work just fine, sysfs nodes are created, but sometimes userspace cannot see the sysfs files that are being created. This is quite odd, because if you look at sysfs, the files are there, but yet, libudev does not think it is.

How to Create a sysfs Files Correctly (cont'd)

It turns out that the driver is racing with userspace to notice when the device is being created in sysfs. The driver core, and at a more basic level, the kobject core below it, announces to userspace when a new device is created or removed from the system. At that point in time, tools run to read all of the attributes for the device, and store them away so that udev rules can run on them, and other libraries can have access to them.

The problem is, when a driver's `probe()` function is called, userspace has already been told the device is present, so any sysfs files that are created at this point in time, will probably be missed entirely.

How to Create a sysfs Files Correctly (cont'd)

The driver core has a number of ways that this can be solved, making the driver author's job even easier, by allowing “default attributes” to be created by the driver core before it is announced to userspace.

These default attribute groups exist at lots of different levels in the driver / device / class hierarchy.

How to Create a sysfs Files Correctly (cont'd)

If you have a bus, you can set the following fields in **struct** `bus_type`:

```
struct bus_type {  
    /* ... */  
    struct bus_attribute    *bus_attrs;  
    struct device_attribute *dev_attrs;  
    struct driver_attribute *drv_attrs;  
    /* ... */  
}
```

If you dealing with a class, you can set the following fields in **struct** `class`:

```
struct class {  
    /* ... */  
    struct class_attribute  *class_attrs;  
    struct device_attribute *dev_attrs;  
    struct bin_attribute    *dev_bin_attrs;  
    /* ... */  
}
```

How to Create a sysfs Files Correctly (cont'd)

If you have control over the **struct** `device_driver` structure, then set this field:

```
struct device_driver {  
    /* ... */  
    const struct attribute_group **groups;  
    /* ... */  
}
```

If you don't have control over the driver either, or want different sysfs files for different devices controlled by your driver. Then, you have control over the device structure itself:




```
struct device {  
    /* ... */  
    const struct attribute_group **groups;  
    /* ... */  
}
```

How to Create a sysfs Files Correctly (cont'd)

By setting this value, you don't have to do anything in your `probe()` or `release()` functions at all in order for the sysfs files to be properly created and destroyed whenever your device is added or removed from the system.

References

References

-  Greg Kroah-Hartman. Everything you never wanted to know about kobjects, ksets, and ktypes
-  Patrick Mochel. The Linux Kernel Driver Model
-  Greg Kroah-Hartman. The Driver Model Core.
-  Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. Linux Device Drivers, 3rd Edition. Ch 14: The Linux Device Model.
-  Robert Love. Linux Kernel Development. Third Edition. 2010. Ch 17: The Device Model.
-  Greg Kroah-Hartman. How to Create a sysfs File Correctly
-  Kernel Recipes 2016 - The Linux Driver Model - Greg KH

Assignments

Assignment

Some user LEDs are installed on the BeagleBone Black PCB and they can be controlled from user space using Linux LED class. Necessary drivers are enablead and configured for BeagleBone Black. Learn about Linux LED class and GPIO LEDs device driver and extend the driver with additional sysfs attribute as described below:

- Learn Documentation:
 - `Documentation/leds/leds-class.txt`;
 - `Documentation/devicetree/bindings/leds/leds-gpio.txt`;
- Find leds-gpio entries in sysfs (try `find /sys -iname "leds-gpio"`);
- Investigate how LED class related data is exposed in sysfs (try `ls -la`, etc.);
- Learn how to control LEDs installed on BeagleBone Black PCB;

Assignment (cont'd)

- Extend GPIO LEDs device driver with new sysfs attribute. Introduce "Do Not Disturb Mode" ("DND Mode") controlled by "dnd-mode" attribute;
 - When DND Mode is enabled (`echo 1 > dnd-mode`) all the LEDs should be turned off and no LEDs should be lighted even if something triggers a LED;
 - When DND Mode is disabled (`echo 0 > dnd-mode`) LEDs behaviour should be switched to default;
- Try to control LEDs when "DND Mode" is enabled and disabled. Ensure your feature works as expected.

Assignment (cont'd)

Report the following results:

- How to turn all the LEDs off using sysfs (default driver, without dnd-mode attribute);
- Prepare and send changes you have made for leds-gpio.c to add dnd-mode attribute (use `git diff`).

Note: it is possible you have break the default GPIO LEDs configuration (DTS, defconfig, etc.) when make kernel changes related to the previous assignments. In this case save you work and try to reset changes (use git) to restore default configuration.

Thank you!