Embedded Linux Device Drivers

Agenda

- Linux Driver Model
- Udev
- Linux Platform Bus

Linux Driver Model

Introduction

- LDM is complex, but it is worth learning.
- Inspired from Windows Driver Model/Foundation (WDM/WDF).
- Linux Driver Model is also calleds as UDM (Unified Driver Model) or udev.
- Added in Linux kernel 2.5.
- Linux kernel 2.6 enforce LDM as default device management system.
- Primary goal of LDM, was proper power management.
- LDM is unification of all the diseparate driver models that were previously used in the kernel.
- In simple words, it enables unified device management.
- https://wiki.linuxfoundation.org/tab/linux-device-driver-model

Objectives

- Device power management: Ordered (sequential) power up and power down of device e.g. devices connected to USB hub.
- Easier user space communication: Enables tight bonding of device and sysfs.
- Hot-plugging: Capability of plugging in/off the device while system is powered on.
- Easier device enumeration: Arranges devices in hierarchial manner (parent-child) to simplify device enumeration.
  - /sys/devices, /sys/bus, /sys/class, /dev/, ...
- Object life cycle: Devices are treated as kernel objects and manage life cycle via reference counting.

Achievements

- Communication with user space becomes very easy via sysfs.
- Enumeration of devices becomes easy due to hierarchial arrangements.
- Linking of device with its device driver and vice-versa become possible.
- Multiple ways to find device due to device categorization.
- Ordered power up/down improve the durability of the device.
- Reference counting mechanism made memory management robust (No memory leakages).
- Code duplication is removed (e.g. Device class/bus code remains common).

Significance

- The Linux device model is a collection of various data structures, and helper functions that provide a unifying and hierarchical view of all the buses, devices, drivers present on the system.
- You can access the whole Linux device and driver model through a pseudo filesystem called sysfs, which is mounted at /sys. Different components of the Linux device model is represented as files and directories through sysfs.
- Sysfs exposes underlying bus, device, and driver details and their relationships in the Linux device model.
- Thus using a various data structures, the Linux device model binds all the buses, the devices, and drivers of the system through its a various data structures.

LDM Structures

- struct kobject
- ktype (struct kobj_type)
- struct kset
- struct subsystem
- struct kref
- struct attribute
- struct device
- struct device_driver
- struct bus_type
- struct class

kobject

- Basic struct for all other struct.
- Present in all device objects.
- Functionalities
  - Device naming/numbering
  - Hierarchial arrangement
  - Type of the device
  - Reference counting
- Include linux/kobject.h
- As it is container data struct i.e. it is not meant for standalone use, but it is used as a member in some outer/container structure.
- The sysfs filesystem gets populated because of kobjects. sysfs is a userspace representation of the kobjects hierarchy inside the kernel.

```
struct kobject {
    const char *name;
    struct list_head entry;
    struct kobject *parent;
    struct kset *kset;
    const struct kobj_type *ktype;
    struct sysfs_dirent *sd;
    struct kref kref;
};
```

```
struct kref {
    atomic_t refcount;
};
```

- kobject functions
  - container_of(): Returns whose container this kobject is.

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

```
static struct cdev cdev;
// ...
void print_cdev(struct kobject *kp) {
    struct cdev *ptr;
    ptr = container_of(kp, struct cdev, kobj);
    // ...
}
// ...
void some_func(void) {
    // ...
    print_cdev(&cdev.kobj);
    // ...
}
```

- kobject_init(): Initalizes kobject (default state/values).
- kobject_cleanup(): Un-initalizes/reclaims kobject.
- kobject_name(): Retrieves name of device container's kobject.
- kobject_set_name(): Set name of device container's kobject.
- kobject_rename(): Renames device container's kobject.
- kobject_get(): Increments the reference count in kobject.
- kobject_put(): Decrements the reference count in kobject.
- kobject_get_path(): Retrieves device path from file-system (hierarchial).

- kobject_hotplug(): Check if device is hotplug enabled or not.
- kobj_map(): Device numbering -- unique hash code
- kobject_add(): Add kobject into the kset.
- kobject_del(): Removes or deletes kobject from the kset.
- kobject_register(): Initalize kobject and adds kobject into the kset.
- kobject_unregister(): Uninitalize kobject and deletes kobject into the kset.

ktype

- Referred as ktype, but struct is kobj_type.
- There is no release memory fuction for kobject, when reference count drops to zero.
- Designer wanted automatic release of memory without calling it explicitly i.e. callback mechanism. Also memory given to the attributes of devices should be released.
- This release() operation is added into another struct - kobj_type. Also it includes the attributes of the device.
- ktype is used to share common attributes/properties among all the kobjects (of its type). Additionally it has common operations in relation with sysfs (e.g. show attr, store attr).
- ktype defines the behaviour of the container object of which kobject is part of.

```
struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
struct attribute {
    char *name;
    struct module *owner;
    mode_t mode;
};
```

kset

- Set of some LDM structs.
- Example:
    - Two kobjects (kobj1 & kobj2) have same attributes, they are grouped in same ktype (ktype1).
    - These two devices (represented by kobj1 & kobj2) are char devices (struct cdev).
    - There are two more kobjects (kobj3 & kobj4) have same attributes (like ktype1), they are also grouped in same ktype (ktype1). But these two devices (represented by kobj3 & kobj4) are block devices (struct block_device).
    - All these four devices can be grouped under same kset (kset1), even though they are different type/catgory (char dev and block dev).
- The kset is group of those kobjects having same ktypes, even if from different class of devices.
- kset is a list of kobjects. Each kobject's parent pointer in a set, points to parent object of the set.

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

- kset functions:
    - kset_init(): Initalize kset to default state.
    - kset_add(): Adds kset into the subsystem.
    - kset_del(): Removes kset from the subsystem.
    - kset_register(): Initalize kset and add into subsystem.
    - kset_unregister(): Uninitalize kset and remove from subsystem.
    - kset_get(): Increments reference count of the kset.
    - kset_put(): Decrements reference count of the kset.
    - kset_find_obj(): Find object of given name in the kset.

subsystem

- Root of LDM hierarchial device management.
- The subsystem is group of related ksets. There can be multiple subsystems.

- Important subsystems are - block, bus, class, device, firmware and module.
- Subsystem contains ksets and synchronization primitives (if devices are used concurrently).

```
struct subsystem {
    struct kset kset;
    struct rw_semaphore rwsem;
};
```

- subsystem functions:
  - decl_subsys(): Macro to declare a new subsystem along with hotplug ops.
  - subsystem_add(): Adds subsystem into sysfs.
  - subsystem_del(): Deletes subsystem into sysfs.
  - subsystem_register(): Initalize subsystem and add into sysfs.
  - subsystem_unregister(): Uninitalize subsystem and remove from sysfs.
  - subsystem_get(): Increments reference count of the subsystem.
  - subsystem_put(): Decrements reference count of the subsystem.

device

- Anything represented by "struct device" is a device in LDM.
- struct device members:
  - struct kobject kobj;
  - struct device *parent;
  - struct bus_type *bus;
  - struct device_driver *driver;
  - void *platform_data; // platform specific data -- not used by device core
  - void *driver_data;
  - struct device_node *of_node; // device tree node
- Mostly devices are specialized i.e. platform_device, i2c_client, etc.

Linux Driver Model

- Kobjects have a name and a reference count. A kobject also has a parent pointer (allowing objects to be arranged into hierarchies), a specific type, and, a representation in the sysfs pseudo filesystem. Kobjects are usually embedded within some other structure which contains the stuff the code is really interested in.
- A ktype is the type of object that embeds a kobject. Every structure that embeds a kobject needs a corresponding ktype. The ktype controls what happens to the kobject when it is created and destroyed.
- A kset is a group of kobjects. These kobjects can be of the same ktype or belong to different ktypes. The kset is the basic container type for collections of kobjects. Ksets contain their own kobjects, but you can safely ignore that implementation detail as the kset core code handles this kobject automatically.
- Kset serves 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."
- 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.
- Kset can support the "hotplugging" of kobjects and influence how uevent events are reported to user space.
- In object-oriented terms, "kset" is the top-level container class.
- A kset keeps its children in a standard kernel linked list. Kobjects point back to their containing kset via their kset field. In almost all cases, the kobjects belonging to a kset have that kset (or, strictly, its embedded kobject) in their parent.
- A sysfs directory full of other directories, generally each of those directories corresponds to a kobject in the same kset.

Udev subsystem

- https://opensource.com/article/18/11/udev
- The udev daemon process (system-udevd) sends events when any device is connected, disconnected, and other events.
- We can write scripts which will be executed on these events. These scripts may have commands/actions like load a device driver, unload a driver, copy the disk, ...
- Important commands
  - sudo udevadm monitor
  - udevadm info -a -n /dev/sda
  - create executable shell scripts -- /usr/local/bin/pendrive-arrival.sh

    ```
    #!/bin/bash
    echo "Pen Drive is Attached: " `date` >> /tmp/pd.log
    ```

  - /etc/udev/rules.d/80-local.rules

        - SUBSYSTEM=="block", ATTRS{idVendor}=="03f0", ACTION=="add", RUN+="/usr/local/bin/pendrive-arrival.sh"
    - sudo udevadm control --reload

Linux platform bus

- Refer slides.

Board Files Example

- https://github.com/beagleboard/linux/blob/master/arch/arm/mach-omap1/board-nokia770.c
    - Refer function: omap_nokia770_init()

Device Tree Example

- Beaglebone-Black -- DTS
    - https://github.com/beagleboard/linux/blob/master/arch/arm/boot/dts/ti/omap/am335x-boneblack.dts
    - https://github.com/beagleboard/linux/blob/master/arch/arm/boot/dts/ti/omap/am335x-boneblack-common.dtsi
    - https://github.com/beagleboard/linux/blob/master/arch/arm/boot/dts/ti/omap/am33xx.dtsi

Assignment

1. Implement a sysfs based driver for BBB GPIO.
    - global variable -- int state;
    - implement sysfs operations -- show and store
        - state_store()
            - state = given_value;
            - if state is 0, gpio_set_value(pin, 0) and if state is not 0, gpio_set_value(pin, 1) -- LED ON/OFF.
        - state_show()
            - display LED state -- 1/0
    - create attribute and attribute group.
    - global variable -- struct kobject led_kobj;
    - module init --
        - set up gpio -- gpio_is_valid(), gpio_request(), gpio_output_direction().

        - kobject_create_and_add()
        - sysfs_create_group()
    - module exit --
        - gpio_free()
        - kobject_put()
    - module macros
2. Optional Assignment 2 -- Implement a sysfs based driver for BBB GPIO to on-off multiple LEDs.
    - Reference: https://elixir.bootlin.com/linux/latest/source/samples/kobject/kobject-example.c