Linux 2.6 Device Model

Vijay Kumar B

<vijaykumar@bravegnu.org>

1. Introduction

In the 2.4 and earlier Linux kernels, there was no unified database of what devices were present in the system, and how they were connected with each other. The implications of this are:

- The user had to grep through log messages to find out if a particular device has been detected by the kernel or not. There was no straight forward method for an application to list out what devices have been detected by the kernel, and whether a driver has been associated with the device.
- It was not possible to do proper power management, because this requires information on how the devices are connected in a system. As an example, before a USB controller is powered down, all the USB peripherals connected to that controller had to be powered down.

To overcome these problems, in 2.5 and later kernels a framework has been provided to maintain a device model. This article describes this device model framework. The intention of this article is to provide a bird's eye view of the working of the device model framework. The specific details of each sub-component can be obtained from various other books/articles and of course the kernel source code.

The five software components that play a major role in building and maintaining the device model are:

- the device model core
- the generic bus drivers
- the bus controller drivers
- the device drivers
- the class drivers

2. Device Model Core

The device model core defines a set of structures and functions. The structures form the building blocks of the device model and the functions update and maintain the device model.

Some of the important structures defined by the device model core are given below.

- `struct bus_type`
- `struct device`
- `struct device_driver`
- `struct class`

The `struct bus_type` is used to represent busses like PCI, USB, I2C, etc. The `struct device` is used to represent devices like an Intel AC97 audio controller, an Intel PRO/100 ethernet controller, a PS/2 mouse etc. The `struct device_driver` is used to represent kernel drivers that can handle specific devices. The `struct class` is used to represent a class of devices like sound, input, graphics, etc. no matter how they are connected to the system.

The device model core, among other things, defines functions to `register` and `unregister` instances of the above structures. These functions are listed below.

- `bus_register()`
- `bus_unregister()`
- `device_register()`
- `device_unregister()`
- `driver_register()`
- `driver_unregister()`
- `class_register()`
- `class_unregister()`

The files that implement the device model core are `include/linux/device.h`, `drivers/base/*.c`.

3. Generic Bus Drivers

For each bus supported by the kernel there is a generic bus driver. The generic bus driver allocates a `struct bus_type` and registers it with the kernel's list of bus types. The registration is done using `bus_register()`. (`bus_type_register()` would have been a more appropriate name!).

The important fields of the bus_type structure are shown below.

```
bus_type
|-- name (string)
|-- !klist_devices (klist)
|-- !klist_drivers (klist)
|-- match (fp)
```

```
|-- suspend (fp)
`-- resume (fp)
```

The fields marked with a `!` are internal to the device model core and should not be touched by the generic bus driver directly.

- The `name` member provides a human readable representation of the bus type, example: pci, usb, mdio.
- The `klist_drivers` member is a list of drivers that can handle devices on that bus. This list is updated by the `driver_register()` which is called when a driver initializes itself.
- The `klist_devices` member is a list of devices in the system that reside on this particular type of bus. This list is updated by `device_register()` which is called when the bus is scanned for devices by the bus controller driver (during initialization or when a gadget is hot plugged.)
- When a new gadget is plugged into the system, the bus controller driver detects the device and calls `device_register()` the list of drivers associated with the bus is iterated over to find out if there are any drivers that can handle the device. The `match` function provided in the `bus_type` structure is used to check if a given driver can handle a given device.
- When a driver module is inserted into the kernel and the driver calls `driver_register()`, the list of devices associated with the bus is iterated over to find out if there are any devices that the driver can handle. The `match` function is used for this purpose.

When a match is found, the device is associated with the device driver. The process of associating a device with a device driver is called binding.

Given below is a sample of bus_type instantiation for the PHY management bus, taken from `drivers/net/phy/mdio_bus.c`.

```
struct bus_type mdio_bus_type = {
        .name           = "mdio_bus",
        .match          = mdio_bus_match,
        .suspend        = mdio_bus_suspend,
        .resume         = mdio_bus_resume,
};
```

Apart from defining a `bus_type`, the generic bus driver defines a bus specific driver structure and a bus specific device structure. These structures extend the generic `struct device_driver` and `struct device` provided by the device model core, by adding bus specific members.

The generic bus driver provides helper functions to register and unregister device drivers that can handle devices on that bus. These helper functions wrap the generic functions provided by the device model core.

4. Bus Controller Drivers

For a specific bus type there could be many different controllers provided by different vendors. Each of these controllers needs a corresponding bus controller driver. The role of a bus controller driver in maintenance of the device model, is similar to that of any other device driver in that, it registers itself with `driver_register()`. But apart from registering itself, it also detects devices on the bus it is controlling and registers the devices on the bus using `device_register()`.

The bus controller driver is responsible for instantiating and registering instances of `struct device` with the device model core. Some of the important members of `struct device` are given below.

```
device
|-- bus_id (string)
|-- bus (bus_type)
|-- parent (device)
`-- !driver (device_driver)
```

The fields marked with a `!` are internal to the device model core and should not be touched by the bus controller driver directly.

- The `bus_id` member is a unique name for the device within a bus type.
- The `bus` member is a pointer to the bus_type to which this device belongs to.
- When a device is registered by the bus controller driver, the `parent` member is pointed to the bus controller device so as to build the physical device tree.
- When a binding occurs and a driver is found that can handle the device, the `driver` member is pointed to the corresponding device driver.

As a sample bus controller driver, the bus controller driver for the PHY management bus on the MPC85xx, is available from `drivers/net/gianfar_mii.c`

5. Device Drivers

Every device driver registers itself with the bus_type using `driver_register()`. After which the device model core tries to bind it with a device. When a device is detected (registered) that can be handled by a particular driver, the `probe` member of the driver is called to instantiate the driver for that particular device.

Each device driver is responsible for instantiating and registering an instance of `struct device_driver` with the device model core. The important members of `struct device_driver` is given below.

```
device_driver
|-- bus (bus_type)
|-- probe (fp)
|-- remove (fp)
|-- suspend (fp)
`-- resume (fp)
```

- The `bus` member is a pointer to the bus_type to which the device driver is registered.
- The `probe` member is a callback function which is called for each device detected that is supported by the driver. The driver should instantiate itself for each device and initialize the device as well.
- The `remove` member is a callback function is called to unbind the driver from the device. This happens when the device is physically removed, when the driver is unloaded, or when the system is shutdown.

As samples of device drivers, see the PHY driver located in `drivers/net/phy/`.

6. Class Drivers

Most users of a system are not bothered about how devices are connected in a system, but what type of devices are connected in the system. A class driver instantiates a `struct class` for the class of devices it represents and registers it with the device model core using `class_register()`. Each device driver is responsible for adding its device to the appropriate class.

The important members of `struct class` is given below.

```
class
|-- name (string)
`-- !devices (list)
```

The fields marked with a `!` mark are internal to the device model core and should not be touched by the class driver directly.

- `name` is the human readable name given to the instance of `struct class` like graphics, sound, etc.
- `devices` is a list of devices that belong to a particular instance of the class. The `devices` list is updated by the device drivers when the instantiate themselves for a device.

7. Conclusion

With these data structures in place, a device tree of how the devices are connected in the system are available, and what types of devices are present in the system is also available. This overcomes the limitations of the 2.4 kernel, and paves way for

- better power management implementations
- better instrospection of devices connected to the system

Last updated 16-Sep-2006 22:18:54 IST