**Linux device Model**

The new Linux Device Model (LDM) is an effort to provide a set of common interfaces for device subsystems to use. 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

**Device Model Core**

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

* struct bus\_type
* struct device
* struct device\_driver :name,probe,remove,suspend & resume function pointers
* 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.

**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()

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.

**Bus controller Driver**

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.

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.

**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 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.

**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.

**Kobject**

Kernel objects are exported as directories via sysfs.

The *kobject* is the fundamental structure that holds the device model together. It was initially conceived as a simple reference counter, but its responsibilities have grown over time, and so have its fields. The tasks handled by struct kobject and its supporting code now include:

*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, underneath it, a kobject that interacts with the kernel to create its visible representation.

*Data structure glue*

The device model is, in its entirety, a fiendishly 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.

A kobject has the type struct kobject; it is defined in *<linux/kobject.h>*. That file also includes declarations for a number of other structures related to kobjects and, of course, a long list of functions for manipulating them.

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.

**Sysfs**

**Sysfs** is a [virtual file system](http://en.wikipedia.org/wiki/Virtual_file_system) provided by [Linux](http://en.wikipedia.org/wiki/Linux_kernel) 2.6. Sysfs exports information about devices and drivers from the kernel device model to userspace, and is also used for configuration. sysfs is a mechanism for representing kernel

objects, their attributes, and their relationships with each other. It provides two components:a kernel programming interface for exporting these items via sysfs, and a user interface to view and manipulate these items that maps back to the kernel objects which they represent.We can mount also

mount -t sysfs sysfs /sys

/sys/

|-- block

|-- bus

|-- class

|-- devices

|-- firmware

|-- module