The Linux Kernel Device Model

Patrick Mochel <mochel@digitalimplant.org>

Drafted 26 August 2002 Updated 31 January 2006

Overview

The Linux Kernel Driver Model is a unification of all the disparate driver models that were previously used in the kernel. It is intended to augment the bus-specific drivers for bridges and devices by consolidating a set of data and operations into globally accessible data structures.

Traditional driver models implemented some sort of tree-like structure (sometimes just a list) for the devices they control. There wasn't any uniformity across the different bus types.

The current driver model provides a common, uniform data model for describing a bus and the devices that can appear under the bus. The unified bus model includes a set of common attributes which all busses carry, and a set of common callbacks, such as device discovery during bus probing, bus shutdown, bus power management, etc.

The common device and bridge interface reflects the goals of the modern computer: namely the ability to do seamless device "plug and play", power management, and hot plug. In particular, the model dictated by Intel and Microsoft (namely ACPI) ensures that almost every device on almost any bus on an x86-compatible system can work within this paradigm. Of course, not every bus is able to support all such operations, although most buses support most of those operations.

Downstream Access

Common data fields have been moved out of individual bus layers into a common data structure. These fields must still be accessed by the bus layers, and sometimes by the device-specific drivers.

Other bus layers are encouraged to do what has been done for the PCI layer. struct pci_dev now looks like this:

```
struct pci_dev {
	...

	struct device dev;	/* Generic device interface */
	...
};
```

Note first that the struct device dev within the struct pci_dev is statically allocated. This means only one allocation on device discovery.

Note also that that struct device dev is not necessarily defined at the front of the pci_dev structure. This is to make people think about what they're doing when switching between the bus driver and the global driver, and to discourage meaningless and incorrect casts between the two.

The PCI bus layer freely accesses the fields of struct device. It knows about the structure of struct pci_dev, and it should know the structure of struct device. Individual PCI device drivers that have been converted to the current driver model generally do not and should not touch the fields of struct device, unless there is a compelling reason to do so.

The above abstraction prevents unnecessary pain during transitional phases. If it were not done this way, then when a field was renamed or removed, every downstream driver would break. On the other hand, if only the bus layer (and not the device layer) accesses the struct device, it is only the bus layer that needs to change.

User Interface

By virtue of having a complete hierarchical view of all the devices in the system, exporting a complete hierarchical view to userspace becomes relatively easy. This has been accomplished by implementing a special purpose virtual file system named sysfs.

Almost all mainstream Linux distros mount this filesystem automatically; you can see some variation of the following in the output of the "mount" command:

```
$ mount
...
none on /sys type sysfs (rw,noexec,nosuid,nodev)
...
$
```

The auto-mounting of sysfs is typically accomplished by an entry similar to the following in the /etc/fstab file:

```
none          /sys     sysfs     defaults               0 0
```

or something similar in the /lib/init/fstab file on Debian-based systems:

```
none            /sys    sysfs    nodev,noexec,nosuid    0 0
```

If sysfs is not automatically mounted, you can always do it manually with:

```
# mount -t sysfs sysfs /sys
```

Whenever a device is inserted into the tree, a directory is created for it. This directory may be populated at each layer of discovery - the global layer, the bus layer, or the device layer.

The global layer currently creates two files - 'name' and 'power'. The former only reports the name of the device. The latter reports the current power state of the device. It will also be used to set the current power state.

The bus layer may also create files for the devices it finds while probing the bus. For example, the PCI layer currently creates 'irq' and 'resource' files for each PCI device.

A device-specific driver may also export files in its directory to expose device-specific data or tunable interfaces.

More information about the sysfs directory layout can be found in the other documents in this directory and in the file Documentation/filesystems/sysfs.rst.