Reprinted from the

Proceedings of the
Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

The Linux Kernel Device Model

Patrick Mochel
Open Source Development Lab
mochel@osdl.org

Abstract

Linux kernel development follows a simple guideline that code should be only as complex as absolutely necessary. This design philosophy has made it easy for thousands of people to contribute code, especially in the realm of device drivers: the kernel supports hundreds of devices on over a dozen peripheral buses.

This bottom-up approach to development has provided a great deal of benefit for users of typical systems in the last decade. However, as Linux progresses into new niches and more requirements are imposed on operating systems of modern hardware, lack of unification among device subsystems poses some serious roadblocks.

The new Linux Device Model (LDM) is an effort to provide a set of common interfaces for device subsystems to use. This foundation is intended to enhance the kernel's support for modern platforms and devices, which require a more unified approach to devices.

This paper discusses the attributes of the LDM and the issues they are designed to resolve. It describes the interfaces in a bottom-up approach; in the same manner in which they were devloped. It also discusses the current progress of the effort, and some potential uses of it in the future.

1 Introduction

The LDM was initially motivated by a single goal: to provide a global device tree that could be used to suspend and resume all devices in a computer during system sleep transitions.

Figure 1 show how all devices in a computer connected. Like devices are grouped on a bus. Buses are linked together via bridge devices. All physical devices can be represented via a single tree structure. This tree structure can be walked to provide proper suspend and resume sequences.

Kernel device subsystems have been developed to concisely represent devices of a particular physical type. Because of this, and because of the vast number of physical configurations possible, there is little data or code shared between subsystems. Figure 2 shows how the PCI device hierarchy is represented internally. Though the PCI tree is physically connected to other devices, this hierarchy is autonomous with regard to other internal device representations.

2 The Linux Device Model Core

In order to construct a global device tree, a common device structure was created to rep-

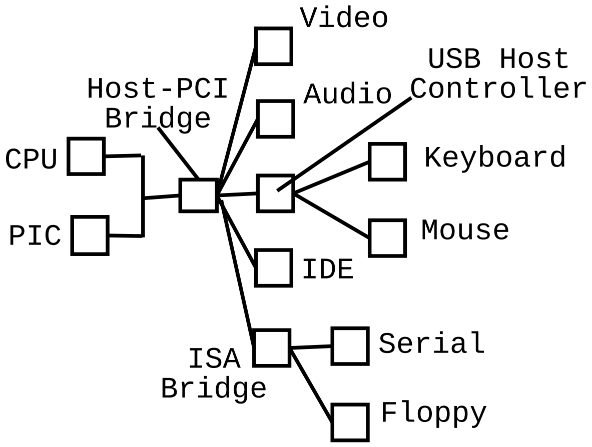

Figure 1: Physical Device Topology

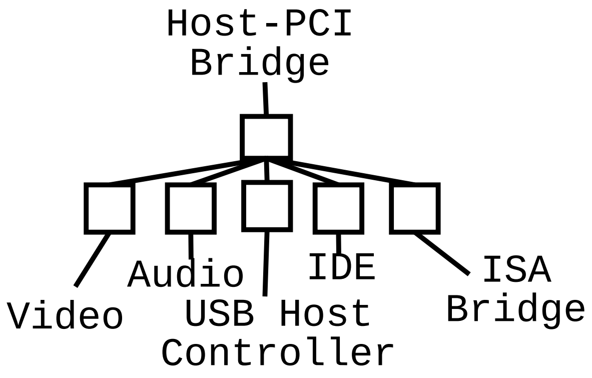

Figure 2: Kernel Repesentation of PCI Topology

```
struct device_driver {
  char   * name;
  list_t   node;
  int (*probe) (struct device * dev);
  int (*remove) (struct device * dev,
                 u32 flags);
  int (*suspend)(struct device * dev,
               u32 state, u32 level);
  int (*resume) (struct device * dev,
                 u32 level);
};

struct device {
  list_t g_list;
  list_t node;
  list_t bus_node;
  list_t children;
  struct device   * parent;

  char    name[DEVICE_NAME_SIZE];
  char    bus_id[BUS_ID_SIZE];

  spinlock_t      lock;
  atomic_t        refcount;

  struct device_driver * driver;
  void                 * driver_data;
};

int device_register(struct device
                    *dev);

/* device reference counting */
void get_device(struct device *dev);
void put_device(struct device *dev);

/* device-level locking */
void lock_device(struct device *dev);
void unlock_device(struct device
                   *dev);
```

Listing 1: The Device Model Core

resent each physical device in the system. Listing 1 includes the definition of `struct device`, which is the minimum set of data necessary to describe each device in the system. It contains little detail about the physical attributes of the device, but provides proper linkage information and support for device-level locking and reference counting.

System bus drivers allocate a device structure for each physical device discovered when probing. The bus driver is responsible for initializing the `bus_id` and `parent` fields of the device and registering the device with the LDM core. The LDM core will then initialize the other fields of the device and add it to the device hierarchy.

Device Reference Counting

The LDM core exports device reference counting primitives

`get_device`, which increments the reference count, and `put_device`, which decrements it. When the reference count reaches 0, it is removed from the device hierarchy and the remove callback of its driver is called to free resources.

The LDM core does not export an interface to explicitly unregister the device. Instead, it relies on reference counting to handle proper garbage collection and removal from the global hierarchy.

The device reference count is initialized to 2 in

`device_register`. It is decremented to 1 when the function exits, leaving the device structure pinned in memory.

Device Locking

The LDM core exports simple primitives to provide device-level locking. The current implementation is a simple spinlock, though this is abstracted from the caller should the type of lock change (e.g. to a semaphore or R/W lock).

Device Drivers

A global device hierarchy allows each device in the system to be represented in a common way. This allows the core to easily walk the device tree to do such things as properly ordered power management transitions. `struct device_driver` in Listing 1 defines a simple set of operations for the core to perform these actions on each device.

The `suspend` and `resume` callbacks provide power management functionality. The `remove` callback is called to logically remove the device from the system. It is called when the device reference count reaches 0, or during system reboot to quiesce all the devices in the system.

`probe` is called when attemptingto bind a driver to a device. This callback is currently unused since driver binding currently happens solely at the bus driver level.

Currently, many bus drivers define a driver similar to this. Instead of converting every device driver to use this common structure, bus drivers implement only one instance of this common structure and bind it to each device discovered. This generic driver then forwards calls to the bus-specific driver. This solution is an interim one only; eventually each driver will use this common structure and register itself with the LDM core instead of a bus.

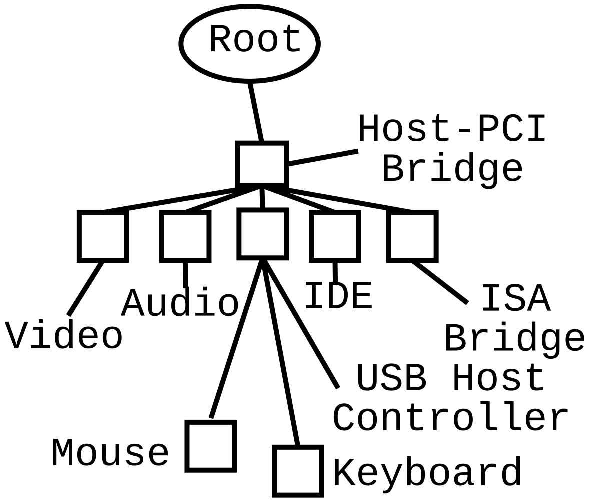

Figure 3: Device Hierarchy with Logical Root Device

3 Completing the Device Tree

The Device Model core was designed to explicitly support the semantics of modern peripheral buses and their drivers, such as PCI and USB. These bus drivers have well-defined and mature methods for discovering devices and representing them locally in a tree-like manner. Because the LDM was based on the existing data and behavior of these bus drivers, converting them to the generic interface typically only involves modifying references to bus-specific structures to generic structures.

There is no common peripheral bus for many of the devices in the system. These devices are referred to as either "platform" devices, including Host-Peripheral Bus bridges and legacy devices; or "system" devices, including CPUs and interrupt controllers. The Linux drivers for these devices represent this logical autonomy.

To complete a global hierarchial representation, these devices must be also be represented. The global hierarchy thus needs some common, top-level entry point.

Device Root

Referring to the figure of device topology, it is apparent that devices are arranged in an acyclical graph, though not necessarily a tree. The kernel bus drivers map subsets of this graph into local tree structures with an explicit root node: the bridge device to the bus. The global hierarchy binds the local trees into one global tree.

Root buses (e.g. root PCI buses) do not have upstream bridges to other peripheral buses. As such, they do not have an explicit parent, and create a forest of devices, instead of one unified tree.

To bind all the devices together, the LDM core creates a logical root device that is the ancestor of all devices in the hierarchy. It is statically allocated and initialized when the LDM core is initialized. Buses that have no obvious parent are registered as children of this device. Figure 3 shows the logical device root and the its relation to the hierarchies of peripheral buses.

Platform Devices

Platform devices are all devices that are physically located on the system board. This includes all legacy devices and host bridges to peripheral buses. host-peripheral bridges are typically not represented in the kernel as devices on a bus; only as parent devices to buses.

These devices appear as autonomous devices in the system responding to I/O requests on hardcoded ports. Drivers for these devices perform device discovery and immediately bind to the devices. These differ from modern bus drivers which perform device discovery in a separate stage than driver binding.

In many modern systems, the system firmware provides information about the devices in the system, often enumerating all of the platform devices. The OS can use this information in lieu of probing legacy I/O ports on platforms that do not support them.To support this firmware enumeration, drivers for platform devices must be taught to use the firmware data for discovery rather than their legacy methods.

Instead of creating special cases in the platform drivers for every firmware discovery mechanism, the method of device discovery is decoupled from the driver binding; legacy probing becomes only one method of device discovery.

```
struct platform_device {
   list_t         node;
   char           name[BUS_ID_SIZE];
   u32            instance;
   struct device  device;
};

int platform_add_device(
   struct device * parent,
   char * busid,
   u32 instance);

struct platform_driver {
   char    * name;
   list_t  node;
};

int platform_register_driver(
      struct platform_driver * drv);
```

Listing 2: The platform bus interface

To implement this, a “platform” bus driver is created to manage platform devices and drivers. As platform devices are discovered, via legacy probing or via a firmware driver, it is added to the bus’s list of devices. As drivers are loaded, they register with the bus, and it attempts to bind them to specific devices. Listing 2 lists the interfaces to the platform bus.

Firmware enumeration usually knows the proper ancestral ordering of the devices, so the device is added in the proper location in the hierarchy. Legacy probing usually does not, though it is not necessary to add any special cases for those devices.

Platform devices are of two types: host-peripheral bridges and legacy devices. Bridges do not have parent devices, so it is valid to pass a NULL parent to `platform_add_device`. Figure 3 displays the logical relationship between the device root and the Host-PCI bridge; `platform_add_device` is the means for representing that relationship in the kernel.

Legacy Devices

Legacy devices usually do have a parent, though it is difficult to infer exactly who it is when legacy probing is used for discovery. Rather than attempt to guess, a logical “legacy bridge” is created to act as surrogate parent for all legacy devices. To register as a legacy device, a driver uses `legacy_add_device`, which internally calls `platform_add_device`, the legacy bridge as the parent.

```
int legacy_add_device(char * busid,
                      u32 instance);
```

Listing 3: Legacy device interface

System Devices

System devices are devices integral to the function of the computer, such as CPUs, APICs, and memory banks. These devices do not follow traditional Unix *read/write* semantics. They do have attributes though, and most have drivers exporting sort of interface to the rest

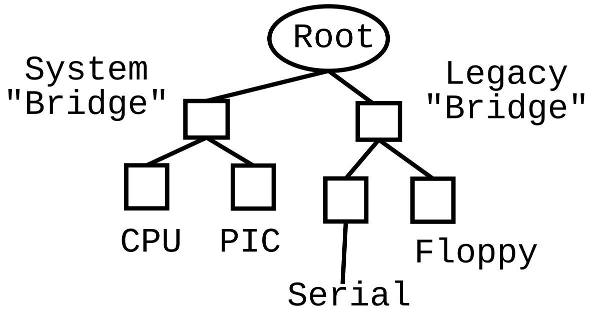

Figure 4: Logical Legacy and System Buses

of the kernel and userspace. However, there are no common bus-level semantics for communicating with the set of system devices as a whole.

It is desirable to group these devices for logical organization. To do this, a logical bus represents the bus that the system devices reside on. Similar to legacy devices, a logical bridge device is created to parent system devices. Devices are added to the system bus using `system_add_device`. Figure 4 displays the hierarchy of the logical buses, and their relationship to the device root.

```
int system_add_device(char * busid,
                      u32 instance);
```

Listing 4: System device interface

4 The User Interface: *driverfs*

During the early development stages of the Device Model, a debugging aid was desired to test various aspects of the code. A device tree, it turns out, maps nicely to a filesystem directory structure.

The device tree was initially exported to userspace using the *procfs* filesystem. A new filesystem, *driverfs*, was soon created to specifically represent the devices. *driverfs* is a simple filesystem based on *ramfs*. It is initialized when the LDM core is initialized, and can be mounted anywhere in filesystem hierarchy.

When registered, every devicehas a *driverfs* directory created on its behalf. It is created in its parent's directory, representing the physical topology. The name of the directory is the `struct device::bus_id field`.

Exporting Device Attributes

The device directories can be populated with files to export device and driver attributes to userspace. These attributes can be accessed using standard the *read* and *write* system calls.

Attributes can be added at any level. The LDM core adds three default files: *name*, *power*, and *status*. Upon device discovery, the bus drivers may add files to export bus-specific attributes. When a driver is bound to a device, it may add files to export device-specific attributes.

Listing 5 shows the `driver_file_entry` object, which is how *driverfs* files are represented. The `show` callback is called when a user reads from a file. The `store` callback is called when a user writes to a file.

To create a file, a caller statically declares a `driver_file_entry` object and initialize the `name`, `mode`, `show` and `store` fields of it. `device_create_file` is used to add the file to the device's directory.

The LDM core clones the `driver_file_entry` structure by allocating a new structure of that type and copying the object passed in. This allows

```
struct driver_file_entry {
   struct driver_dir_entry * parent;
   struct list_head          node;
   char                    * name;
   mode_t                    mode;
   struct dentry           * dentry;

   ssize_t (*show)(struct device
             * dev,
             char * buf,
             size_t count,
             loff_t off);
   ssize_t (*store)(struct device
             * dev,
             const char * buf,
             size_t count,
             loff_t off);
};

int
device_create_file(struct device
  *device,
  struct driver_file_entry * entry);
```

Listing 5: driverfs interface

the caller to reuse the same file description to create files for multiple devices without having to manually allocate and initialize each instance.

Operation

The *driverfs* core stores a pointer to the `driver_file_entry` structure in the private data fields of the VFS objects representing the file. From this pointer, the *driverfs* core can obtain the pointer to the device structure. This pointer is then referenced on read and write operations.

During a read operation, *driverfs* allocates a page-sized buffer and passes the buffer pointer to it to the `show` callback. The driver fills the buffer and returns. It is then copied to userspace.

When the file is written to, *driverfs* allocates another page-sized buffer and fills it with data copied from userspace. It passes the buffer pointer to the `store` callback of the driver, which consumes the data.

File Format

The preferred contents of *driverfs* files is one ASCII-encoded value per file. Although these preferences are not enforced, maintaining this standard has several usability advantages:

- A user can read from and write to the file using **cat** and **echo**; tools found on any Linux distribution.
- Coordination between kernel drivers and user space consumers becomes easier; there is no proprietary format for each file.
- A file's contents are obvious to a command line user.
- A file's contents are obvious to a programmer looking at the driver source code.
- It eases creation of automated tools to export device attributes in a more user-friendly manner (i.e. via a GUI).

5 Bus Drivers

At the time of this writing, most LDM development is concentrated on creating a generic bus type object and set of operations to operate on this type. Listing 6 defines a structure to wrap attributes common across all buses. Consolidating bus data affords the creation of generic routines to manipulate that data.

```
struct bus_type {
       char       * name;
       list_t     node;
       spinlock_t lock;
       list_t     devices;
       list_t     drivers;
};

int bus_register(
    struct bus_type * bus);
void bus_unregister(
    struct bus_type * bus);

int driver_register(
        struct device_driver * drv);
void driver_unregister(
        struct device_driver * drv);
```

Listing 6: Bus driver interface

Bus drivers typically maintain a list of all devices on all buses of their type. This allows for easy searches of devices when binding drivers. Insertion into this list can happen when the bus driver calls `device_register()` for a device.

Device drivers register with their bus, which insert the driver into an internal list and attempt to bind it with every present device. Drivers can instead be taught to use only the generic `struct device_driver` and register with the LDM core.This would insert the driver in the bus's list of drivers, then attempt to bind the driver to the devices on that particular bus (by calling the driver's probe callback for each device).

Device insertion at runtime requires registering a device with the bus and attempting to bind a known driver to it. A userspace agent (**/sbin/hotplug**) is executed to finish configuring the device.

Hotplug insertion events are nearly identical to device discovery when a bus is initially probed, though no buses attempt to bind drivers to devices when they are initially discovered. Driver binding can be coupled to device discovery when the bus is probed, making all device discoveries appear as hotplug events. With centralized lists to manage devices and drivers, this binding can take place when the device is registered with the LDM core.

Many buses do not do locking on their internal lists or reference counting on their devices and drivers. By centralizing the list manipulation routines, proper locking and reference counting can be guaranteed for all buses.

driverfs exports an accurate physical representation of the device hierarchy. It is difficult to navigate, though, since devices can be buried under several obscure directories. By centrally managing bus lists, devices and drivers can easily be added to a driverfs directory owned by the bus.

Conclusion

The Linux Device Model is an effort to consolidate data and interfaces from the many disparate device and driver models in the Linux kernel today. It allows the kernel to do things never possible before, like proper power management and shutdown sequences. It provides common infrastructure to guarantee proper locking, reference counting, and handling of hotplug events for all bus types.