Porting Drivers to the New Driver Model

Patrick Mochel

7 January 2003

Overview

Please refer to Documentation/driver-model/*.txt for definitions of
various driver types and concepts.

Most of the work of porting devices drivers to the new model happens
at the bus driver layer. This was intentional, to minimize the
negative effect on kernel drivers, and to allow a gradual transition
of bus drivers.

In a nutshell, the driver model consists of a set of objects that can
be embedded in larger, bus-specific objects. Fields in these generic
objects can replace fields in the bus-specific objects.

The generic objects must be registered with the driver model core. By
doing so, they will exported via the sysfs filesystem. sysfs can be
mounted by doing

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

The Process

Step 0: Read include/linux/device.h for object and function definitions.

Step 1: Registering the bus driver.

- Define a struct bus_type for the bus driver.

```
struct bus_type pci_bus_type = {
        .name           = "pci",
};
```

- Register the bus type.
  This should be done in the initialization function for the bus type,
  which is usually the module_init(), or equivalent, function.

```
static int __init pci_driver_init(void)
{
        return bus_register(&pci_bus_type);
}

subsys_initcall(pci_driver_init);
```

The bus type may be unregistered (if the bus driver may be compiled as a module) by doing:

```
bus_unregister(&pci_bus_type);
```

- Export the bus type for others to use.

Other code may wish to reference the bus type, so declare it in a shared header file and export the symbol.

From include/linux/pci.h:

```
extern struct bus_type pci_bus_type;
```

From file the above code appears in:

```
EXPORT_SYMBOL(pci_bus_type);
```

- This will cause the bus to show up in /sys/bus/pci/ with two subdirectories: 'devices' and 'drivers'.

```
# tree -d /sys/bus/pci/
/sys/bus/pci/
|-- devices
`-- drivers
```

Step 2: Registering Devices.

struct device represents a single device. It mainly contains metadata describing the relationship the device has to other entities.

- Embed a struct device in the bus-specific device type.

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

It is recommended that the generic device not be the first item in the struct to discourage programmers from doing mindless casts between the object types. Instead macros, or inline functions, should be created to convert from the generic object type.

```
#define to_pci_dev(n) container_of(n, struct pci_dev, dev)
```

or

```
static inline struct pci_dev * to_pci_dev(struct kobject * kobj)
{
        return container_of(n, struct pci_dev, dev);
}
```

  This allows the compiler to verify type-safety of the operations
  that are performed (which is Good).

- Initialize the device on registration.

  When devices are discovered or registered with the bus type, the
  bus driver should initialize the generic device. The most important
  things to initialize are the bus_id, parent, and bus fields.

  The bus_id is an ASCII string that contains the device's address on
  the bus. The format of this string is bus-specific. This is
  necessary for representing devices in sysfs.

  parent is the physical parent of the device. It is important that
  the bus driver sets this field correctly.

  The driver model maintains an ordered list of devices that it uses
  for power management. This list must be in order to guarantee that
  devices are shutdown before their physical parents, and vice versa.
  The order of this list is determined by the parent of registered
  devices.

  Also, the location of the device's sysfs directory depends on a
  device's parent. sysfs exports a directory structure that mirrors
  the device hierarchy. Accurately setting the parent guarantees that
  sysfs will accurately represent the hierarchy.

  The device's bus field is a pointer to the bus type the device
  belongs to. This should be set to the bus_type that was declared
  and initialized before.

  Optionally, the bus driver may set the device's name and release
  fields.

  The name field is an ASCII string describing the device, like

     "ATI Technologies Inc Radeon QD"

  The release field is a callback that the driver model core calls
  when the device has been removed, and all references to it have
  been released. More on this in a moment.

- Register the device.

  Once the generic device has been initialized, it can be registered
  with the driver model core by doing:

       device_register(&dev->dev);

It can later be unregistered by doing:

```
device_unregister(&dev->dev);
```

This should happen on buses that support hotpluggable devices.
If a bus driver unregisters a device, it should not immediately free
it. It should instead wait for the driver model core to call the
device's release method, then free the bus-specific object.
(There may be other code that is currently referencing the device
structure, and it would be rude to free the device while that is
happening).

When the device is registered, a directory in sysfs is created.
The PCI tree in sysfs looks like:

```
/sys/devices/pci0/
|-- 00:00.0
|-- 00:01.0
|   `-- 01:00.0
|-- 00:02.0
|   `-- 02:1f.0
|       `-- 03:00.0
|-- 00:1e.0
|   `-- 04:04.0
|-- 00:1f.0
|-- 00:1f.1
|   |-- ide0
|   |   |-- 0.0
|   |   `-- 0.1
|   `-- ide1
|       `-- 1.0
|-- 00:1f.2
|-- 00:1f.3
`-- 00:1f.5
```

Also, symlinks are created in the bus's 'devices' directory
that point to the device's directory in the physical hierarchy.

```
/sys/bus/pci/devices/
|-- 00:00.0 -> ../../../devices/pci0/00:00.0
|-- 00:01.0 -> ../../../devices/pci0/00:01.0
|-- 00:02.0 -> ../../../devices/pci0/00:02.0
|-- 00:1e.0 -> ../../../devices/pci0/00:1e.0
|-- 00:1f.0 -> ../../../devices/pci0/00:1f.0
|-- 00:1f.1 -> ../../../devices/pci0/00:1f.1
|-- 00:1f.2 -> ../../../devices/pci0/00:1f.2
|-- 00:1f.3 -> ../../../devices/pci0/00:1f.3
|-- 00:1f.5 -> ../../../devices/pci0/00:1f.5
|-- 01:00.0 -> ../../../devices/pci0/00:01.0/01:00.0
|-- 02:1f.0 -> ../../../devices/pci0/00:02.0/02:1f.0
|-- 03:00.0 -> ../../../devices/pci0/00:02.0/02:1f.0/03:00.0
`-- 04:04.0 -> ../../../devices/pci0/00:1e.0/04:04.0
```

Step 3: Registering Drivers.

struct device_driver is a simple driver structure that contains a set
of operations that the driver model core may call.

- Embed a struct device_driver in the bus-specific driver.

  Just like with devices, do something like:

```
struct pci_driver {
       ...
       struct device_driver    driver;
};
```

- Initialize the generic driver structure.

  When the driver registers with the bus (e.g. doing pci_register_driver()),
  initialize the necessary fields of the driver: the name and bus
  fields.

- Register the driver.

  After the generic driver has been initialized, call

```
       driver_register(&drv->driver);
```

  to register the driver with the core.

  When the driver is unregistered from the bus, unregister it from the
  core by doing:

```
       driver_unregister(&drv->driver);
```

  Note that this will block until all references to the driver have
  gone away. Normally, there will not be any.

- Sysfs representation.

  Drivers are exported via sysfs in their bus's 'driver's directory.
  For example:

```
/sys/bus/pci/drivers/
|-- 3c59x
|-- Ensoniq AudioPCI
|-- agpgart-amdk7
|-- e100
`-- serial
```

Step 4: Define Generic Methods for Drivers.

struct device_driver defines a set of operations that the driver model core calls. Most of these operations are probably similar to operations the bus already defines for drivers, but taking different parameters.

It would be difficult and tedious to force every driver on a bus to simultaneously convert their drivers to generic format. Instead, the bus driver should define single instances of the generic methods that forward call to the bus-specific drivers. For instance:

```
static int pci_device_remove(struct device * dev)
{
        struct pci_dev * pci_dev = to_pci_dev(dev);
        struct pci_driver * drv = pci_dev->driver;

        if (drv) {
                if (drv->remove)
                        drv->remove(pci_dev);
                pci_dev->driver = NULL;
        }
        return 0;
}
```

The generic driver should be initialized with these methods before it is registered.

```
        /* initialize common driver fields */
        drv->driver.name = drv->name;
        drv->driver.bus = &pci_bus_type;
        drv->driver.probe = pci_device_probe;
        drv->driver.resume = pci_device_resume;
        drv->driver.suspend = pci_device_suspend;
        drv->driver.remove = pci_device_remove;

        /* register with core */
        driver_register(&drv->driver);
```

Ideally, the bus should only initialize the fields if they are not already set. This allows the drivers to implement their own generic methods.

Step 5: Support generic driver binding.

The model assumes that a device or driver can be dynamically registered with the bus at any time. When registration happens, devices must be bound to a driver, or drivers must be bound to all devices that it supports.

A driver typically contains a list of device IDs that it supports. The bus driver compares these IDs to the IDs of devices registered with it. The format of the device IDs, and the semantics for comparing them are bus-specific, so the generic model does attempt to generalize them.

Instead, a bus may supply a method in struct bus_type that does the comparison:

```
  int (*match)(struct device * dev, struct device_driver * drv);
```

match should return '1' if the driver supports the device, and '0' otherwise.

When a device is registered, the bus's list of drivers is iterated over. bus->match() is called for each one until a match is found.

When a driver is registered, the bus's list of devices is iterated over. bus->match() is called for each device that is not already claimed by a driver.

When a device is successfully bound to a driver, device->driver is set, the device is added to a per-driver list of devices, and a symlink is created in the driver's sysfs directory that points to the device's physical directory:

```
/sys/bus/pci/drivers/
|-- 3c59x
|   `-- 00:0b.0 -> ../../../../devices/pci0/00:0b.0
|-- Ensoniq AudioPCI
|-- agpgart-amdk7
|   `-- 00:00.0 -> ../../../../devices/pci0/00:00.0
|-- e100
|   `-- 00:0c.0 -> ../../../../devices/pci0/00:0c.0
`-- serial
```

This driver binding should replace the existing driver binding mechanism the bus currently uses.

Step 6: Supply a hotplug callback.

Whenever a device is registered with the driver model core, the userspace program /sbin/hotplug is called to notify userspace. Users can define actions to perform when a device is inserted or removed.

The driver model core passes several arguments to userspace via environment variables, including

- ACTION: set to 'add' or 'remove'
- DEVPATH: set to the device's physical path in sysfs.

A bus driver may also supply additional parameters for userspace to consume. To do this, a bus must implement the 'hotplug' method in struct bus_type:

```
     int (*hotplug) (struct device *dev, char **envp,
                     int num_envp, char *buffer, int buffer_size);
```

This is called immediately before /sbin/hotplug is executed.

Step 7: Cleaning up the bus driver.

The generic bus, device, and driver structures provide several fields
that can replace those defined privately to the bus driver.

- Device list.

struct bus_type contains a list of all devices registered with the bus
type. This includes all devices on all instances of that bus type.
An internal list that the bus uses may be removed, in favor of using
this one.

The core provides an iterator to access these devices.

```
int bus_for_each_dev(struct bus_type * bus, struct device * start,
                     void * data, int (*fn)(struct device *, void *));
```


- Driver list.

struct bus_type also contains a list of all drivers registered with
it. An internal list of drivers that the bus driver maintains may
be removed in favor of using the generic one.

The drivers may be iterated over, like devices:

```
int bus_for_each_drv(struct bus_type * bus, struct device_driver * start,
                     void * data, int (*fn)(struct device_driver *, void *));
```


Please see drivers/base/bus.c for more information.

- rwsem

struct bus_type contains an rwsem that protects all core accesses to
the device and driver lists. This can be used by the bus driver
internally, and should be used when accessing the device or driver
lists the bus maintains.

- Device and driver fields.

Some of the fields in struct device and struct device_driver duplicate
fields in the bus-specific representations of these objects. Feel free
to remove the bus-specific ones and favor the generic ones. Note
though, that this will likely mean fixing up all the drivers that
reference the bus-specific fields (though those should all be 1-line
changes).