Device Drivers

```
struct device_driver {
        char                    * name;
        struct bus_type         * bus;

        struct completion       unloaded;
        struct kobject          kobj;
        list_t                  devices;

        struct module           *owner;

        int     (*probe)        (struct device * dev);
        int     (*remove)       (struct device * dev);

        int     (*suspend)      (struct device * dev, pm_message_t state);
        int     (*resume)       (struct device * dev);
};
```

Allocation
~~~~~~~~~~

Device drivers are statically allocated structures. Though there may be multiple devices in a system that a driver supports, struct device_driver represents the driver as a whole (not a particular device instance).

Initialization
~~~~~~~~~~~~~~

The driver must initialize at least the name and bus fields. It should also initialize the devclass field (when it arrives), so it may obtain the proper linkage internally. It should also initialize as many of the callbacks as possible, though each is optional.

Declaration
~~~~~~~~~~~

As stated above, struct device_driver objects are statically allocated. Below is an example declaration of the eepro100 driver. This declaration is hypothetical only; it relies on the driver being converted completely to the new model.

```
static struct device_driver eepro100_driver = {
       .name            = "eepro100",
       .bus             = &pci_bus_type,

       .probe           = eepro100_probe,
       .remove          = eepro100_remove,
       .suspend         = eepro100_suspend,
       .resume          = eepro100_resume,
};
```
~~~~~~~~~~~

Most drivers will not be able to be converted completely to the new
model because the bus they belong to has a bus-specific structure with
bus-specific fields that cannot be generalized.

The most common example of this are device ID structures. A driver
typically defines an array of device IDs that it supports. The format
of these structures and the semantics for comparing device IDs are
completely bus-specific. Defining them as bus-specific entities would
sacrifice type-safety, so we keep bus-specific structures around.

Bus-specific drivers should include a generic struct device_driver in
the definition of the bus-specific driver. Like this:

```
struct pci_driver {
       const struct pci_device_id *id_table;
       struct device_driver       driver;
};
```

A definition that included bus-specific fields would look like
(using the eepro100 driver again):

```
static struct pci_driver eepro100_driver = {
       .id_table       = eepro100_pci_tbl,
       .driver         = {
                .name           = "eepro100",
                .bus            = &pci_bus_type,
                .probe          = eepro100_probe,
                .remove         = eepro100_remove,
                .suspend        = eepro100_suspend,
                .resume         = eepro100_resume,
       },
};
```

Some may find the syntax of embedded struct initialization awkward or
even a bit ugly. So far, it's the best way we've found to do what we want...

Registration
~~~~~~~~~~~~

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

The driver registers the structure on startup. For drivers that have
no bus-specific fields (i.e. don't have a bus-specific driver
structure), they would use driver_register and pass a pointer to their
struct device_driver object.

Most drivers, however, will have a bus-specific structure and will
need to register with the bus using something like pci_driver_register.

It is important that drivers register their driver structure as early as
possible. Registration with the core initializes several fields in the
struct device_driver object, including the reference count and the
lock. These fields are assumed to be valid at all times and may be
used by the device model core or the bus driver.
~~~~~~~~~~~~

Transition Bus Drivers

By defining wrapper functions, the transition to the new model can be made easier. Drivers can ignore the generic structure altogether and let the bus wrapper fill in the fields. For the callbacks, the bus can define generic callbacks that forward the call to the bus-specific callbacks of the drivers.

This solution is intended to be only temporary. In order to get class information in the driver, the drivers must be modified anyway. Since converting drivers to the new model should reduce some infrastructural complexity and code size, it is recommended that they are converted as class information is added.

Access

Once the object has been registered, it may access the common fields of the object, like the lock and the list of devices.

```
int driver_for_each_dev(struct device_driver * drv, void * data,
                        int (*callback)(struct device * dev, void * data));
```

The devices field is a list of all the devices that have been bound to the driver. The LDM core provides a helper function to operate on all the devices a driver controls. This helper locks the driver on each node access, and does proper reference counting on each device as it accesses it.

sysfs

When a driver is registered, a sysfs directory is created in its bus's directory. In this directory, the driver can export an interface to userspace to control operation of the driver on a global basis; e.g. toggling debugging output in the driver.

A future feature of this directory will be a 'devices' directory. This directory will contain symlinks to the directories of devices it supports.

Callbacks

```
        int     (*probe)        (struct device * dev);
```

The probe() entry is called in task context, with the bus's rwsem locked and the driver partially bound to the device. Drivers commonly use container_of() to convert "dev" to a bus-specific type, both in probe() and other routines. That type often provides device resource data, such as pci_dev.resource[] or platform_device.resources, which is used in addition to dev->platform_data to initialize the driver.

This callback holds the driver-specific logic to bind the driver to a
given device. That includes verifying that the device is present, that
it's a version the driver can handle, that driver data structures can
be allocated and initialized, and that any hardware can be initialized.
Drivers often store a pointer to their state with dev_set_drvdata().
When the driver has successfully bound itself to that device, then probe()
returns zero and the driver model code will finish its part of binding
the driver to that device.

A driver's probe() may return a negative errno value to indicate that
the driver did not bind to this device, in which case it should have
released all resources it allocated.

```
        int     (*remove)       (struct device * dev);
```

remove is called to unbind a driver from a device. This may be
called if a device is physically removed from the system, if the
driver module is being unloaded, during a reboot sequence, or
in other cases.

It is up to the driver to determine if the device is present or
not. It should free any resources allocated specifically for the
device; i.e. anything in the device's driver_data field.

If the device is still present, it should quiesce the device and place
it into a supported low-power state.

```
        int     (*suspend)      (struct device * dev, pm_message_t state);
```

suspend is called to put the device in a low power state.

```
        int     (*resume)       (struct device * dev);
```

Resume is used to bring a device back from a low power state.

Attributes
~~~~~~~~~~

```
struct driver_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device_driver *driver, char *buf);
        ssize_t (*store)(struct device_driver *, const char * buf, size_t
count);
};
```

Device drivers can export attributes via their sysfs directories.
Drivers can declare attributes using a DRIVER_ATTR macro that works
identically to the DEVICE_ATTR macro.

Example:

```
DRIVER_ATTR(debug,0644,show_debug,store_debug);
```

This is equivalent to declaring:
~~~~~~~~~~

```
struct driver_attribute driver_attr_debug;
```

This can then be used to add and remove the attribute from the
driver's directory using:

```
int driver_create_file(struct device_driver *, const struct driver_attribute *);
void driver_remove_file(struct device_driver *, const struct driver_attribute
*);
```