The Basic Device Structure
~~~~~~~~~~~~~~~~~~~~~~~~~~

```
struct device {
        struct list_head g_list;
        struct list_head node;
        struct list_head bus_list;
        struct list_head driver_list;
        struct list_head intf_list;
        struct list_head children;
        struct device   * parent;

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

        spinlock_t      lock;
        atomic_t        refcount;

        struct bus_type * bus;
        struct driver_dir_entry dir;

        u32             class_num;

        struct device_driver *driver;
        void            *driver_data;
        void            *platform_data;

        u32             current_state;
        unsigned char *saved_state;

        void    (*release)(struct device * dev);
};
```

Fields
~~~~~~

g_list: Node in the global device list.

node:   Node in device's parent's children list.

bus_list: Node in device's bus's devices list.

driver_list:   Node in device's driver's devices list.

intf_list:     List of intf_data. There is one structure allocated for each interface that the device supports.

children:      List of child devices.

parent:        *** FIXME ***

name:          ASCII description of device.
               Example: " 3Com Corporation 3c905 100BaseTX [Boomerang]"

bus_id:        ASCII representation of device's bus position. This field should be a name unique across all devices on the bus type the device belongs to.

Example: PCI bus_ids are in the form of
<bus number>:<slot number>.<function number>
This name is unique across all PCI devices in the system.

lock: Spinlock for the device.

refcount: Reference count on the device.

bus: Pointer to struct bus_type that device belongs to.

dir: Device's sysfs directory.

class_num: Class-enumerated value of the device.

driver: Pointer to struct device_driver that controls the device.

driver_data: Driver-specific data.

platform_data: Platform data specific to the device.

Example: for devices on custom boards, as typical of embedded and SOC based hardware, Linux often uses platform_data to point to board-specific structures describing devices and how they are wired. That can include what ports are available, chip variants, which GPIO pins act in what additional roles, and so on. This shrinks the "Board Support Packages" (BSPs) and minimizes board-specific #ifdefs in drivers.

current_state: Current power state of the device.

saved_state: Pointer to saved state of the device. This is usable by the device driver controlling the device.

release: Callback to free the device after all references have gone away. This should be set by the allocator of the device (i.e. the bus driver that discovered the device).

## Programming Interface

The bus driver that discovers the device uses this to register the device with the core:

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

The bus should initialize the following fields:

- parent
- name
- bus_id
- bus

A device is removed from the core when its reference count goes to 0. The reference count can be adjusted using:

```
struct device * get_device(struct device * dev);
void put_device(struct device * dev);
```

get_device() will return a pointer to the struct device passed to it
if the reference is not already 0 (if it's in the process of being
removed already).

A driver can access the lock in the device structure using:

```
void lock_device(struct device * dev);
void unlock_device(struct device * dev);
```

## Attributes

```
struct device_attribute {
	struct attribute	attr;
	ssize_t (*show)(struct device *dev, struct device_attribute *attr,
			char *buf);
	ssize_t (*store)(struct device *dev, struct device_attribute *attr,
			 const char *buf, size_t count);
};
```

Attributes of devices can be exported via drivers using a simple
procfs-like interface.

Please see Documentation/filesystems/sysfs.txt for more information
on how sysfs works.

Attributes are declared using a macro called DEVICE_ATTR:

```
#define DEVICE_ATTR(name,mode,show,store)
```

Example:

```
DEVICE_ATTR(power,0644,show_power,store_power);
```

This declares a structure of type struct device_attribute named
'dev_attr_power'. This can then be added and removed to the device's
directory using:

```
int device_create_file(struct device *device, struct device_attribute * entry);
void device_remove_file(struct device * dev, struct device_attribute * attr);
```

Example:

```
device_create_file(dev,&dev_attr_power);
device_remove_file(dev,&dev_attr_power);
```

The file name will be 'power' with a mode of 0644 (-rw-r--r--).

Word of warning:  While the kernel allows device_create_file() and
device_remove_file() to be called on a device at any time, userspace has
strict expectations on when attributes get created.  When a new device is
registered in the kernel, a uevent is generated to notify userspace (like

udev) that a new device is available. If attributes are added after the device is registered, then userspace won't get notified and userspace will not know about the new attributes.

This is important for device driver that need to publish additional attributes for a device at driver probe time. If the device driver simply calls device_create_file() on the device structure passed to it, then userspace will never be notified of the new attributes. Instead, it should probably use class_create() and class->dev_attrs to set up a list of desired attributes in the modules_init function, and then in the .probe() hook, and then use device_create() to create a new device as a child of the probed device. The new device will generate a new uevent and properly advertise the new attributes to userspace.

For example, if a driver wanted to add the following attributes:

```
struct device_attribute mydriver_attribs[] = {
        __ATTR(port_count, 0444, port_count_show),
        __ATTR(serial_number, 0444, serial_number_show),
        NULL
};
```

Then in the module init function is would do:

```
        mydriver_class = class_create(THIS_MODULE, "my_attrs");
        mydriver_class.dev_attr = mydriver_attribs;
```

And assuming 'dev' is the struct device passed into the probe hook, the driver probe function would do something like:

```
        create_device(&mydriver_class, dev, chrdev, &private_data, "my_name");
```