Bus Types

Definition

See the kerneldoc for the struct bus_type.

int bus_register(struct bus_type * bus);

Declaration

Each bus type in the kernel (PCI, USB, etc) should declare one static object of this type. They must initialize the name field, and may optionally initialize the match callback:

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

The structure should be exported to drivers in a header file:

extern struct bus_type pci_bus_type;

Registration

When a bus driver is initialized, it calls bus_register. This initializes the rest of the fields in the bus object and inserts it into a global list of bus types. Once the bus object is registered, the fields in it are usable by the bus driver.

Callbacks

match(): Attaching Drivers to Devices

The format of device ID structures and the semantics for comparing them are inherently bus-specific. Drivers typically declare an array of device IDs of devices they support that reside in a bus-specific driver structure.

The purpose of the match callback is to give the bus an opportunity to determine if a particular driver supports a particular device by comparing the device IDs the driver supports with the device ID of a particular device, without sacrificing bus-specific functionality or type-safety.

When a driver is registered with the bus, the bus's list of devices is iterated over, and the match callback is called for each device that does not have a driver associated with it.

Device and Driver Lists

The lists of devices and drivers are intended to replace the local lists that many buses keep. They are lists of struct devices and struct device_drivers, respectively. Bus drivers are free to use the lists as they please, but conversion to the bus-specific type may be necessary.

The LDM core provides helper functions for iterating over each list:

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

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

These helpers iterate over the respective list, and call the callback for each device or driver in the list. All list accesses are synchronized by taking the bus's lock (read currently). The reference count on each object in the list is incremented before the callback is called; it is decremented after the next object has been obtained. The lock is not held when calling the callback.

sysfs

There is a top-level directory named 'bus'.

Each bus gets a directory in the bus directory, along with two default directories:

```
/sys/bus/pci/
|-- devices
`-- drivers
```

Drivers registered with the bus get a directory in the bus's drivers directory:

```
/sys/bus/pci/
|-- devices
`-- drivers
    |-- Intel ICH
    |-- Intel ICH Joystick
    |-- agpgart
    `-- e100
```

Each device that is discovered on a bus of that type gets a symlink in the bus's devices directory to the device's directory in the physical hierarchy:

```
/sys/bus/pci/
|-- devices
|   |-- 00:00.0 -> ../../../root/pci0/00:00.0
|   |-- 00:01.0 -> ../../../root/pci0/00:01.0
|   `-- 00:02.0 -> ../../../root/pci0/00:02.0
`-- drivers
```

Exporting Attributes

```
struct bus_attribute {
      struct attribute        attr;
      ssize_t (*show)(struct bus_type *, char * buf);
      ssize_t (*store)(struct bus_type *, const char * buf, size_t count);
};
```

Bus drivers can export attributes using the BUS_ATTR_RW macro that works similarly to the DEVICE_ATTR_RW macro for devices. For example, a definition like this:

```
static BUS_ATTR_RW(debug);
```

is equivalent to declaring:

```
static bus_attribute bus_attr_debug;
```

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

```
int bus_create_file(struct bus_type *, struct bus_attribute *);
void bus_remove_file(struct bus_type *, struct bus_attribute *);
```