Platform Devices and Drivers

See <linux/platform_device.h> for the driver model interface to the platform bus: platform_device, and platform_driver. This pseudo-bus is used to connect devices on busses with minimal infrastructure, like those used to integrate peripherals on many system-on-chip processors, or some "legacy" PC interconnects; as opposed to large formally specified ones like PCI or USB.

Platform devices

Platform devices are devices that typically appear as autonomous entities in the system. This includes legacy port-based devices and host bridges to peripheral buses, and most controllers integrated into system-on-chip platforms. What they usually have in common is direct addressing from a CPU bus. Rarely, a platform_device will be connected through a segment of some other kind of bus; but its registers will still be directly addressable.

Platform devices are given a name, used in driver binding, and a list of resources such as addresses and IRQs:

```
struct platform_device {
      const char      *name;
      u32             id;
      struct device   dev;
      u32             num_resources;
      struct resource *resource;
};
```

Platform drivers

Platform drivers follow the standard driver model convention, where discovery/enumeration is handled outside the drivers, and drivers provide probe() and remove() methods. They support power management and shutdown notifications using the standard conventions:

```
struct platform_driver {
      int (*probe)(struct platform_device *);
      int (*remove)(struct platform_device *);
      void (*shutdown)(struct platform_device *);
      int (*suspend)(struct platform_device *, pm_message_t state);
      int (*suspend_late)(struct platform_device *, pm_message_t state);
      int (*resume_early)(struct platform_device *);
      int (*resume)(struct platform_device *);
      struct device_driver driver;
};
```

Note that probe() should in general verify that the specified device hardware actually exists; sometimes platform setup code can't be sure. The probing can use device resources, including clocks, and device platform_data.

Platform drivers register themselves the normal way:

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

Or, in common situations where the device is known not to be hot-pluggable, the probe() routine can live in an init section to reduce the driver's runtime memory footprint:

```
int platform_driver_probe(struct platform_driver *drv,
                  int (*probe)(struct platform_device *))
```

Kernel modules can be composed of several platform drivers. The platform core provides helpers to register and unregister an array of drivers:

```
int __platform_register_drivers(struct platform_driver * const *drivers,
                                unsigned int count, struct module *owner);
void platform_unregister_drivers(struct platform_driver * const *drivers,
                                 unsigned int count);
```

If one of the drivers fails to register, all drivers registered up to that point will be unregistered in reverse order. Note that there is a convenience macro that passes THIS_MODULE as owner parameter:

```
#define platform_register_drivers(drivers, count)
```

Device Enumeration

As a rule, platform specific (and often board-specific) setup code will register platform devices:

```
int platform_device_register(struct platform_device *pdev);

int platform_add_devices(struct platform_device **pdevs, int ndev);
```

The general rule is to register only those devices that actually exist, but in some cases extra devices might be registered. For example, a kernel might be configured to work with an external network adapter that might not be populated on all boards, or likewise to work with an integrated controller that some boards might not hook up to any peripherals.

In some cases, boot firmware will export tables describing the devices that are populated on a given board. Without such tables, often the only way for system setup code to set up the correct devices is to build a kernel for a specific target board. Such board-specific kernels are common with embedded and custom systems development.

In many cases, the memory and IRQ resources associated with the platform device are not enough to let the device's driver work. Board setup code will often provide additional information using the device's platform_data field to hold additional information.

Embedded systems frequently need one or more clocks for platform devices, which are normally kept off until they're actively needed (to save power). System setup also associates those clocks with the device, so that calls to clk_get(&pdev->dev, clock_name) return them as needed.

Legacy Drivers: Device Probing

Some drivers are not fully converted to the driver model, because they take on a non-driver role: the driver registers its platform device, rather than leaving that for system infrastructure. Such drivers can't be hotplugged or coldplugged, since those mechanisms require device creation to be in a different system component than the driver.

The only "good" reason for this is to handle older system designs which, like original IBM PCs, rely on error-prone "probe-the-hardware" models for hardware configuration. Newer systems have largely abandoned that model, in favor of bus-level support for dynamic configuration (PCI, USB), or device tables provided by the boot firmware (e.g. PNPACPI on x86). There are too many conflicting options about what might be where, and even educated guesses by an operating system will be wrong often enough to make trouble.

This style of driver is discouraged. If you're updating such a driver, please try to move the device enumeration to a more appropriate location, outside the driver. This will usually be cleanup, since such drivers tend to already have "normal" modes, such as ones using device nodes that were created by PNP or by platform device setup.

None the less, there are some APIs to support such legacy drivers. Avoid using these calls except with such hotplug-deficient drivers:

```
struct platform_device *platform_device_alloc(
                const char *name, int id);
```

You can use platform_device_alloc() to dynamically allocate a device, which you will then initialize with resources and platform_device_register(). A better solution is usually:

```
struct platform_device *platform_device_register_simple(
                const char *name, int id,
                struct resource *res, unsigned int nres);
```

You can use platform_device_register_simple() as a one-step call to allocate and register a device.

Device Naming and Driver Binding

The platform_device.dev.bus_id is the canonical name for the devices. It's built from two components:

- platform_device.name ... which is also used to for driver matching.
- platform_device.id ... the device instance number, or else "-1" to indicate there's only one.

These are concatenated, so name/id "serial"/0 indicates bus_id "serial.0", and "serial/3" indicates bus_id "serial.3"; both would use the platform_driver named "serial". While "my_rtc"/-1 would be bus_id "my_rtc" (no instance id) and use the platform_driver called "my_rtc".

Driver binding is performed automatically by the driver core, invoking driver probe() after finding a match between device and driver. If the probe() succeeds, the driver and device are bound as usual. There are three different ways to find such a match:

- Whenever a device is registered, the drivers for that bus are checked for matches. Platform devices should be registered very early during system boot.
- When a driver is registered using platform_driver_register(), all unbound devices on that bus are checked for matches. Drivers usually register later during booting, or by module loading.
- Registering a driver using platform_driver_probe() works just like using platform_driver_register(), except that the driver won't be probed later if another device registers. (Which is OK, since this interface is only for use with non-hotpluggable devices.)

Early Platform Devices and Drivers

The early platform interfaces provide platform data to platform device drivers early on during the system boot. The code is built on top of the early_param() command line parsing and can be executed very early on.

Example: "earlyprintk" class early serial console in 6 steps

1. Registering early platform device data

The architecture code registers platform device data using the function early_platform_add_devices(). In the case of early serial console this should be hardware configuration for the serial port. Devices registered at this point will later on be matched against early platform drivers.

2. Parsing kernel command line

The architecture code calls parse_early_param() to parse the kernel command line. This will execute all matching early_param() callbacks. User specified early platform devices will be registered at this point. For the early serial console case the user can specify port on the kernel command line as "earlyprintk=serial.0" where "earlyprintk" is the class string, "serial" is the name of the platform driver and 0 is the platform device id. If the id is -1 then the dot and the id can be omitted.

3. Installing early platform drivers belonging to a certain class

The architecture code may optionally force registration of all early platform drivers belonging to a certain class using the function early_platform_driver_register_all(). User specified devices from step 2 have priority over these. This step is omitted by the serial driver example since the early serial driver code should be disabled unless the user has specified port on the kernel command line.

4. Early platform driver registration

Compiled-in platform drivers making use of early_platform_init() are automatically registered during step 2 or 3. The serial driver example should use early_platform_init("earlyprintk", &platform_driver).

5. Probing of early platform drivers belonging to a certain class

The architecture code calls early_platform_driver_probe() to match registered early platform devices associated with a certain class with registered early platform drivers. Matched devices will get probed(). This step can be executed at any point during the early boot. As soon as possible may be good for the serial port case.

6. Inside the early platform driver probe()

The driver code needs to take special care during early boot, especially when it comes to memory allocation and interrupt registration. The code in the probe() function can use is_early_platform_device() to check if it is called at early platform device or at the regular platform device time. The early serial driver performs register_console() at this point.

For further information, see <linux/platform_device.h>.