

[patch 2.6.19-rc6] Documentation/driver-model/platform.txt update/rewrite

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-11/msg05299.html>

- *From:* David Brownell <david-b@xxxxxxxxxxxx>
 - *Date:* Thu, 16 Nov 2006 23:30:14 -0800
-

This is almost a rewrite of the driver-model/platform.txt documentation; the previous text was obsolete (for several years), evidently it never got updated to match the change from being a PC "legacy_bus" to the more widely used core bus for most embedded systems.

Signed-off-by: David Brownell <dbrownell@xxxxxxxxxxxxxxxxxxxxxxxx>

This presumes the previous patch, adding platform_driver_probe().

Index: g26/Documentation/driver-model/platform.txt

--- g26.orig/Documentation/driver-model/platform.txt 2006-11-12 12:22:31.000000000 -0800
+++ g26/Documentation/driver-model/platform.txt 2006-11-16 18:18:16.000000000 -0800
@@ -1,99 +1,131 @@

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.
+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 addressible.

+

+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

~~~~~

-Drivers for platform devices are typically very simple and  
-unstructured. Either the device was present at a particular I/O port  
-and the driver was loaded, or it was not. There was no possibility  
-of hotplugging or alternative discovery besides probing at a specific  
-I/O address and expecting a specific response.

-  
-

-Other Architectures, Modern Firmware, and new Platforms

~~~~~

-These devices are not always at the legacy I/O ports. This is true on
-other architectures and on some modern architectures. In most cases,
-the drivers are modified to discover the devices at other well-known
-ports for the given platform. However, the firmware in these systems
-does usually know where exactly these devices reside, and in some
-cases, it's the only way of discovering them.

-
-

-The Platform Bus

~~~~~

-A platform bus has been created to deal with these issues. First and  
-foremost, it groups all the legacy devices under a common bus, and  
-gives them a common parent if they don't already have one.

-  
-

-But, besides the organizational benefits, the platform bus can also  
-accommodate firmware-based enumeration.

-  
-

-Device Discovery

~~~~~

-The platform bus has no concept of probing for devices. Devices
-discovery is left up to either the legacy drivers or the
-firmware. These entities are expected to notify the platform of
-devices that it discovers via the bus's add() callback:

-
-

- platform_bus.add(parent,bus_id).

-
-

-Bus IDs

~~~~~

-Bus IDs are the canonical names for the devices. There is no globally standard addressing mechanism for legacy devices. In the IA-32 world, we have Pnp IDs to use, as well as the legacy I/O ports. However, neither tell what the device really is or have any meaning on other platforms.

-

-Since both PnP IDs and the legacy I/O ports (and other standard I/O ports for specific devices) have a 1:1 mapping, we map the platform-specific name or identifier to a generic name (at least within the scope of the kernel).

-

-For example, a serial driver might find a device at I/O 0x3f8. The ACPI firmware might also discover a device with PnP ID (\_HID) PNP0501. Both correspond to the same device and should be mapped to the canonical name 'serial'.

-

-The bus\_id field should be a concatenation of the canonical name and the instance of that type of device. For example, the device at I/O port 0x3f8 should have a bus\_id of "serial0". This places the responsibility of enumerating devices of a particular type up to the discovery mechanism. But, they are the entity that should know best (as opposed to the platform bus driver).

-

-Drivers

~~~~~

-Drivers for platform devices should have a name that is the same as the canonical name of the devices they support. This allows the platform bus driver to do simple matching with the basic data structures to determine if a driver supports a certain device.

-

-For example, a legacy serial driver should have a name of 'serial' and register itself with the platform bus.

-

-

-Driver Binding

~~~~~

-Legacy drivers assume they are bound to the device once they start up and probe an I/O port. Divorcing them from this will be a difficult process. However, that shouldn't prevent us from implementing firmware-based enumeration.

-

-The firmware should notify the platform bus about devices before the legacy drivers have had a chance to load. Once the drivers are loaded, the driver model core will attempt to bind the driver to any previously-discovered devices. Once that has happened, it will be free to discover any other devices it pleases.

+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 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 *))
```

+

+

+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 that  
+calls to clk\_get(&pdev->dev, clock\_name) return them as needed.  
+  
+  
+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 catenated, 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 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.)  
+  
-  
To unsubscribe from this list: send the line "unsubscribe linux-kernel" in  
the body of a message to majordomo@xxxxxxxxxxxxxxxxx  
More majordomo info at <http://vger.kernel.org/majordomo-info.html>  
Please read the FAQ at <http://www.tux.org/lkml/>