The platform device API

By **Jonathan Corbet**
June 21, 2011

In the very early days, Linux users often had to tell the kernel where specific devices were to be found before their systems would work. In the absence of this information, the driver could not know which I/O ports and interrupt line(s) the device was configured to use. Happily, we now live in the days of busses like PCI which have discoverability built into them; any device sitting on a PCI bus can tell the system what sort of device it is and where its resources are. So the kernel can, at boot time, enumerate the devices available and everything Just Works.

Alas, life is not so simple; there are plenty of devices which are still not discoverable by the CPU. In the embedded and system-on-chip world, non-discoverable devices are, if anything, increasing in number. So the kernel still needs to provide ways to be told about the hardware that is actually present. "Platform devices" have long been used in this role in the kernel. This article will describe the interface for platform devices; it is meant as needed background material for a following article on integration with device trees.

Platform drivers

A platform device is represented by `struct platform_device`, which, like the rest of the relevant declarations, can be found in `<linux/platform_device.h>`. These devices are deemed to be connected to a virtual "platform bus"; drivers of platform devices must thus register themselves as such with the platform bus code. This registration is done by way of a `platform_driver` structure:

```
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 (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
};
```

At a minimum, the `probe()` and `remove()` callbacks must be supplied; the other callbacks have to do with power management and should be provided if they are relevant.

The other thing the driver must provide is a way for the bus code to bind actual devices to the driver; there are two mechanisms which can be used for that purpose. The first is the `id_table` argument; the relevant structure

is:

```
struct platform_device_id {
    char name[PLATFORM_NAME_SIZE];
    kernel_ulong_t driver_data;
};
```

If an ID table is present, the platform bus code will scan through it every time it has to find a driver for a new platform device. If the device's name matches the name in an ID table entry, the device will be given to the driver for management; a pointer to the matching ID table entry will be made available to the driver as well. As it happens, though, most platform drivers do not provide an ID table at all; they simply provide a name for the driver itself in the `driver` field. As an example, the i2c-gpio driver turns two GPIO lines into an i2c bus; it sets itself up as a platform device with:

```
static struct platform_driver i2c_gpio_driver = {
    .driver         = {
            .name   = "i2c-gpio",
            .owner  = THIS_MODULE,
    },
    .probe          = i2c_gpio_probe,
    .remove         = __devexit_p(i2c_gpio_remove),
};
```

With this setup, any device identifying itself as "`i2c-gpio`" will be bound to this driver; no ID table is needed.

Platform drivers make themselves known to the kernel with:

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

As soon as this call succeeds, the driver's `probe()` function can be called with new devices. That function gets as an argument a `platform_device` pointer describing the device to be instantiated:

```
struct platform_device {
    const char      *name;
    int             id;
    struct device   dev;
    u32             num_resources;
    struct resource *resource;
    const struct platform_device_id *id_entry;
    /* Others omitted */
};
```

The `dev` structure can be used in contexts where it is needed - the DMA mapping API, for example. If the device was matched using an ID table entry, `id_entry` will point to the specific entry matched. The `resource` array can be used to learn where various resources, including memory-mapped I/O registers and interrupt lines, can be found. There are a number of helper functions for getting data out of the resource array; these include:

```
struct resource *platform_get_resource(struct platform_device *pdev,
                                       unsigned int type, unsigned int n);
struct resource *platform_get_resource_byname(struct platform_device *pdev,
                                       unsigned int type, const char *name);
int platform_get_irq(struct platform_device *pdev, unsigned int n);
```

The "`n`" parameter says which resource of that type is desired, with zero indicating the first one. Thus, for example, a driver could find its second MMIO region with:

```
r = platform_get_resource(pdev, IORESOURCE_MEM, 1);
```

Assuming the `probe()` function finds the information it needs, it should verify the device's existence to the extent possible, register the "real" devices associated with the platform device, and return zero.

Platform devices

So now we have a driver for a platform device, but no actual devices yet. As was noted at the beginning, platform devices are inherently not discoverable, so there must be another way to tell the kernel about their existence. That is typically done with the creation of a static `platform_device` structure providing, at a minimum, a name which is used to find the associated driver. So, for example, a simple (fictional) device might be set up this way:

```
static struct resource foomatic_resources[] = {
    {
            .start  = 0x10000000,
            .end    = 0x10001000,
            .flags  = IORESOURCE_MEM,
            .name   = "io-memory"
    },
    {
            .start  = 20,
            .end    = 20,
            .flags  = IORESOURCE_IRQ,
            .name   = "irq",
    }
};

static struct platform_device my_foomatic = {
    .name           = "foomatic",
    .resource       = foomatic_resources,
    .num_resources  = ARRAY_SIZE(foomatic_resources),
};
```

These declarations describe a "`foomatic`" device with a one-page MMIO region starting at `0x10000000` and using IRQ 20. The device is made known to the system with:

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

Once both a platform device and an associated driver have been registered, the driver's `probe()` function will be called and the device will be instantiated. Registration of device and driver are usually done in different places and can happen in either order. A call to `platform_device_unregister()` can be used to remove a platform device.

Platform data

The above information is adequate to instantiate a simple platform device, but many devices are more complex than that. Even the simple i2c-gpio driver described above needs two additional pieces of information: the numbers of the GPIO lines to be used as i2c clock and data lines. The mechanism used to pass this information is called "platform data"; in short, one defines a structure containing the specific information needed and passes it in the platform device's `dev.platform_data` field.

With the i2c-gpio example, a full configuration looks like this:

```
#include <linux/i2c-gpio.h>

static struct i2c_gpio_platform_data my_i2c_plat_data = {
```

```
        .scl_pin        = 100,
        .sda_pin        = 101,
};

static struct platform_device my_gpio_i2c = {
        .name           = "i2c-gpio",
        .id             = 0,
        .dev = {
                .platform_data = &my_i2c_plat_data,
        }
};
```

When the driver's `probe()` function is called, it can fetch the `platform_data` pointer and use it to obtain the rest of the information it needs.

Not everybody in the kernel community is enamored with platform devices; they seem like a bit of a hack used to encode information about specific hardware platforms into the kernel. Additionally, the platform data mechanism lacks any sort of type checking; drivers must simply assume that they have been passed a structure of the expected type. Even so, platform devices are heavily used, and that's unlikely to change, though the means by which they are created and discovered is changing. The way of the future appears to be device trees, which will be described in the following article.

The platform device API
Posted Jun 23, 2011 7:26 UTC (Thu) by **Lumag** (subscriber, #22579) [Link]

Just a small addition: for lots of use cases it's better not to register static instance of platform device, but rather allocate it dynamically, via platform_device_register_{simple,data,resndata}() functions or via platform_device_alloc()/platform_device_add().

The platform device API
Posted Jun 23, 2011 23:47 UTC (Thu) by **dlang** (subscriber, #313) [Link]

I remember having to do some editing of source to get all the serial ports working in the early kernels (ISA bus ports with oddball interrupts and ports to fit a bunch of ports into one machine)

passive voice
Posted Jun 24, 2011 18:31 UTC (Fri) by **giraffedata** (subscriber, #1954) [Link]

> Once both a platform device and an associated driver have been registered, the driver's probe() function will be called and the device will be instantiated. Registration of device and driver are usually done in different places and can happen in either order. A call to platform_device_unregister() can be used to remove a platform device.

Here's an excellent example of the evils of passive voice (and its cousin, nominalization - using a noun instead of a verb for an action). It is not immediately clear who is doing all these things (registering, calling, instantiating, using). Were the paragraph in active voice, that information would be unmissable.

Plus there's the fact that the human brain is set up to comprehend things by forming an image of A acting on B, rather than of some abstract action by abstract agents taking place in the ether. So a reader would have to do considerably less processing to comprehend this paragraph in active voice.

Reply to this comment

passive voice
Posted Jun 28, 2011 12:13 UTC (Tue) by **nye** (guest, #51576) [Link]

>Plus there's the fact that the human brain is set up to comprehend things by forming an image of A acting on B, rather than of some abstract action by abstract agents taking place in the ether. So a reader would have to do considerably less processing to comprehend this paragraph in active voice.

[Citation Needed]

I don't disagree that this particular example could be improved as you say, but your generalization from that seems suspect. It's entirely dependent on whether the agent or the patient is the point of interest.

(I've noticed the passive voice is strongly discouraged by American style guides, even in cases where it is clearly the better choice, whereas it's more widely accepted in the UK for some reason. Given this, it's possible that familiarity has some effect on the brain's ability to comprehend.)

Reply to this comment

passive voice
Posted Jun 28, 2011 15:37 UTC (Tue) by **giraffedata** (subscriber, #1954) [Link]

> > Plus there's the fact that the human brain is set up to comprehend things by forming an image of A acting on B, rather than of some abstract action by abstract agents taking place in the ether. So a reader would have to do considerably less processing to comprehend this paragraph in active voice.
>
> I don't disagree that this particular example could be improved as you say, but your generalization from that seems suspect.

You may be confusing multiple arguments, because I did not generalize from this example to my statement about psychology. I got the generalization from a technical writing class so long ago I can't remember, taught by a technical communications researcher. From that, I specialized to criticism of this particular sentence.

> It's entirely dependent on whether the agent or the patient is the point of interest. ... it's possible that familiarity has some effect on the brain's ability to comprehend

I don't see the connection. It doesn't seem to matter whether you're focusing on A or B in the proposition that it's easier to comprehend A acting on B than the abstract action. (E.g. John throwing a ball as opposed to the abstract concept of the throwing of a ball).

But I admit it is possible to write passive voice which actually does describe A acting on B ("the ball was thrown by John"), even though often the very reason the writer used passive voice was to avoid identifying the agent. Where the agent is so exposed, the only thing left to argue about is whether the brain starts processing "thrown" before integrating John into the picture. And that might depend upon familiarity with

> the syntax.

Reply to this comment