Platform Devices and Drivers

Introducing the Linux Device Model API

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

The Linux Kernel's Device Model

An abstraction for managing devices:

- What they are, where they are
- How they're related to each other
- What services they offer to users
- How devices respond to state changes

The Linux Kernel's Device Model

Facilitates:

- Power management
- Orderly system startup and shutdown
- Communications with user space
- System configurability
- Smarter user applications

The Linux Kernel's Device Model

It's big!

- Lots of code
- LOTS of data structures

Overwhelming, at first:

- Solves a complex problem

The Linux Kernel's Device Model

But:

- You don't need to understand how it works!
- You just need to know how to use it well!

The keys:

- Proper terminology!
- Correct schemas!

The Linux Kernel's Device Model

"Split" implementation model:

- "Driver" is separated from "device"
- Analogous to class "methods" and "data"
- Promotes reuse, portability
- See `struct device_driver` and `struct device`

The Linux Kernel's Device Model

Examples of "devices":

- A literal, physical chip
- Something plugged into a USB port
- Distinct functionalities within a physical chip
- Abstractions e.g. organization, API consistency

Platform Devices and Drivers

Platform devices:

- Wrapper around the full device model API
- System-on-chip peripherals
- Memory-mapped devices
- Minimal bus infrastructures

See `struct platform_device`

Platform Devices and Drivers

Ideal device model entry point:

- Makes some simplifying assumptions
- Hides lots of details

Other kinds of devices:

- I2C
- SPI
- USB
- ...

`struct platform_device`

Represents a “device”:

- Physical, or sometimes virtual
- Often used for SoC, MMIO peripherals
- Unique `name:id` per device

```
#include <linux/platform_device.h>
```

struct platform_device

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

`struct platform_device`

`.name`

- Name of associated driver
- (Devices don't have names!)

`.id`

- Unique numeric device identifier

`.dev`

- Underlying `struct device`

struct platform_device

.num_resources

- Number of resources utilized by the device

.resources

- List of device resources

struct platform_device

.id_entry

- List of compatible drivers, when a multitude exists
- Allows for more generic drivers
- (Relatively new addition to the API)

We will return to this later

`struct platform_driver`

Driver for a platform device:

- Matched by `.name` with devices
- Serves *all* devices of the given name

A true “device driver” in the Linux sense!

struct platform_driver

```
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;
};
```

struct platform_driver

```
struct device_driver {
   ...
   const char     *name;
   ...
};
```

struct platform_driver

`.probe()`

- Invoked when device and driver are associated
- Association occurs at device or driver registration

`.remove()`

- Invoked when device or driver are withdrawn

`struct platform_driver`

"Where is `open()`, etc.?"

- Those aren't device driver methods under Linux!
- See "device attributes"
- See `struct cdev` and `struct miscdevice`

We will return to this topic later

Registering Platform Devices and Drivers

Nominal device use case:

- Allocate a static `struct platform_device`
- Fill out `.name`, `.id`, etc. fields
- Pass to `platform_device_register()`

Usually during board initialization

Registering Platform Devices and Drivers

Nominal driver use case:

- Allocate a static `struct platform_driver`
- Fill out `.driver.name`, `.probe`, etc.
- Pass to `platform_driver_register()`

Usually during module initialization

platform-device.c

```
1  #include <linux/module.h>
2  #include <linux/platform_device.h>
```

```
1  static struct platform_device pex = {
2      .name = "platform-example",
3      .id = -1,
4      .dev = {
5          .platform_data = &pex_platform_data,
6      },
7  };
```

platform-device.c

```
1  static int __init platform_example_init(void)
2  {
3      return platform_device_register(&pex);
4  }
5
6  static void __exit platform_example_exit(void)
7  {
8      platform_device_unregister(&pex);
9  }
```

platform-device.c

```
1   struct pex_platform_data {
2       int gpio_number;
3       int irq_number;
4   };
5
6   static __init struct pex_platform_data
7   pex_platform_data = {
8       .gpio_number = PEX_GPIO,
9       .irq_number = PEX_IRQ,
10  };
```

`platform-device-alloc.c`

The previous example doesn't actually work:

- Statically-allocated devices aren't unpluggable!
- (They lack a `.release()` method)

platform-device-alloc.c

```
1
2   static int __init platform_example_init(void)
3   {
4       pex = platform_device_alloc("platform-example", -1);
5       if (IS_ERR_OR_NULL(pex)) {
6           if (!pex)
7               return -ENOMEM;
8           return PTR_ERR(pex);
9       }
10      return platform_device_add(pex);
```

platform-driver.c

```
1   static int probe(struct platform_device *p)
2   {
3       dev_err(&p->dev, "%s\n", __FUNCTION__);
4       return 0;
5   }
6
7   static int __devexit remove(struct platform_device *p)
8   {
9       dev_err(&p->dev, "%s\n", __FUNCTION__);
10      return 0;
11  }
```

platform-driver.c

```
1  static struct platform_driver pex = {
2      .probe          = probe,
3      .remove         = __devexit_p(remove),
4      .driver           = {
5          .name           = "platform-example",
6          .owner          = THIS_MODULE,
7      },
8  };
```

platform-driver.c

```
1  static int __init platform_example_init(void)
2  {
3      return platform_driver_register(&pex);
4  }
5
6  static void __exit platform_example_exit(void)
7  {
8      platform_driver_unregister(&pex);
9  }
10
11 module_init(platform_example_init);
12 module_exit(platform_example_exit);
```

platform-driver.c

```
1  MODULE_LICENSE("GPL");
2  MODULE_VERSION("0.0");
3  MODULE_AUTHOR("Bill Gatliff <bgat@billgatliff.com>");
4  MODULE_DESCRIPTION("Example platform driver");
```

`dev_get_drvdata()` and `dev_set_drvdata()`

Device-specific data:

- Buffers, etc.
- Completions, mutexes, etc. etc.

dev_get_drvdata() and dev_set_drvdata()

```
1   #include <linux/module.h>
2   #include <linux/slab.h>
3   #include <linux/platform_device.h>
4   #include <linux/pex.h> /* for struct pex_data */
5
6   struct pex {
7       int x;
8       int y;
9       int z;
10  };
```

dev_get_drvdata() and dev_set_drvdata()

```
1   static int probe(struct platform_device *p)
2   {
3       struct pex *pex = kzalloc(sizeof(*pex), GFP_KERNEL);
4       struct pex_data *pex_data = p->dev.platform_data;
5
6       if (!pex)
7           return -ENOMEM;
8
9       platform_set_drvdata(p, pex);
10
11      return 0;
12  }
```

struct resource

```
struct resource {
  resource_size_t start;
  resource_size_t end;
  const char     *name;
  unsigned long  flags;
  struct resource *parent,
                  *sibling,
                  *child;
};
```

struct resource

`IORESOURCE_IO`

- `start`, `end` refer to I/O ports

`IORESOURCE_MEM`

- `start`, `end` refer to memory addresses

`IORESOURCE_IRQ`

- `start`, `end` refer to interrupt channels

struct resource

```
struct resource pxa27x_ohci_resources[] = {
  [0] = {
    .start  = 0x4C000000,
    .end    = 0x4C00ff6f,
    .flags  = IORESOURCE_MEM,
  },
  [1] = {
    .start  = IRQ_USBH1,
    .end    = IRQ_USBH1,
    .flags  = IORESOURCE_IRQ,
  },
};
```

struct resource

```
struct platform_device pxa27x_device_ohci = {
  .name            = "pxa27x-ohci",
  .id              = -1,
  .num_resources   = ARRAY_SIZE(pxa27x_ohci_resources),
  .resource        = pxa27x_ohci_resources,
};
```

struct resource

```
struct resource *r;

r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
if (!r) {
    pr_err(``no resource of IORESOURCE_MEM'');
    retval = -ENXIO;
    goto err1;
}
```

struct resource

```
hcd->rsrc_start = r->start;
hcd->rsrc_len = resource_size(r);

if (!request_mem_region(hcd->rsrc_start,
                        hcd->rsrc_len, hcd_name)) {
    pr_debug(``request_mem_region failed'');
    retval = -EBUSY;
    goto err1;
  }
  ...
```

Recap

Platform devices:

- Represent physical or virtual system components
- `struct platform_device`

Platform drivers:

- Drivers for platform devices
- `struct platform_driver`

Recap

The “platform” concept:

- Thin veneer over device model API
- Simplifies some device model details
- Associates devices and drivers
- Used in non-self-identifying busses

Platform Devices and Drivers

Introducing the Linux Device Model API

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer