

# 第 12 章

## Linux 设备驱动的软件架构思想

### 本章导读

在前面几章我们看到了 globalmem、globalfifo 这样类型的简单的字符设备驱动，但是，纵观 Linux 内核的源代码，读者几乎找不到有如此简单形式的驱动。

在实际的 Linux 驱动中，Linux 内核尽量做得更多，以便于底层的驱动可以做得更少。而且，也特别强调了驱动的跨平台特性。因此，Linux 内核势必会为不同的驱动子系统设计不同的框架。

12.1 节从总体上分析了 Linux 驱动的软件架构设计的出发点。

12.2 以 platform 为例，全面介绍了 platform 设备和驱动，以及 platform 的意义。从而讲明了 Linux 内核中驱动和设备分离的优点。

12.3 节以 RTC、Framebuffer、input、tty、混杂设备驱动等为例，讲解了驱动分层、核心层与底层交互的一般方法。

12.4 节分析了 Linux 设备驱动中主机与外设驱动分离的设计思想，并以 SPI 主机和外设驱动为例进行了佐证。

### 12.1 Linux 驱动的软件架构

Linux 不是为了某单一电路板而设计的操作系统，它可以支持约 30 种体系结构下一定数量的硬件，因此，它的驱动架构很显然不能像 RTOS 下或者无操作系统下那么小儿科的做法。

Linux 设备驱动非常重视软件的可重用和跨平台能力。譬如，如果我们写下一个 DM9000 网卡的驱动，Linux 的想法是这个驱动应该最好一行都不要改就可以在任何一个平台上跑起来。为了做到这一点（看似很难，因为每个板子连接 DM9000 的基址，中断号什么的都可能不一样），驱动中势必会有类似这样的代码：

```
#ifdef BOARD_XXX  
#define DM9000_BASE 0x10000  
#define DM9000_IRQ 8  
#elif defined(BOARD_YYY)
```

```
#define DM9000_BASE 0x20000
#define DM9000_IRQ 7
#elif defined(BOARD_ZZZ)
#define DM9000_BASE 0x30000
#define DM9000_IRQ 9
...
#endif
```

上述代码主要有如下问题：

1) 此段代码看起来面目可憎，如果有 100 个板子，就要 if/else 100 次，到了第 101 个板子，又得重新加 if/else。代码进行着简单的“复制—粘贴”，“复制—粘贴”式的简单重复通常意味着代码编写者的水平很差。

2) 非常难做到一个驱动支持多个设备，如果某个电路板上有两个 DM9000 网卡，则 DM9000\_BASE 这个宏就不够用了，此时势必要定义出来 DM9000\_BASE 1、DM9000\_BASE 2、DM9000\_IRQ 1、DM9000\_IRQ 2 类的宏；定义了 DM9000\_BASE 1、DM9000\_BASE 2 后，如果有第 3 个 DM9000 网卡加到板子上，前面的代码就又不适用了。

3) 依赖于 make menuconfig 选择的项目来编译内核，因此，在不同的硬件平台下要依赖于所选择的 BOARD\_XXX、BOARD\_YYY 选项来决定代码逻辑。这不符合 ARM Linux 3.x 一个映像适用于多个硬件的目标。实际上，我们可能同时选择了 BOARD\_XXX、BOARD\_YYY、BOARD\_ZZZ。

我们按照上面的方法编写代码的时候，相信自己编着编着也会觉得奇怪，闻到了代码里不好的味道。这个时候，请停下你飞奔的脚步，等一等你的灵魂。我们有没有办法把设备端的信息从驱动里面剥离出来，让驱动以某种标准方法拿到这些平台信息呢？Linux 总线、设备和驱动模型实际上可以做到这一点，驱动只管驱动，设备只管设备，总线则负责匹配设备和驱动，而驱动则以标准途径拿到板级信息，这样，驱动就可以放之四海而皆准了，如图 12.1 所示。

Linux 的字符设备驱动需要编写 file\_operations 成员函数，并负责处理阻塞、非阻塞、多路复用、SIGIO 等复杂事物。但是，当我们面对一个真实的硬件驱动时，假如要编写一个按键的驱动，作为一个“懒惰”的程序员，你真的只想做最简单的工作，譬如，收到一个按键中断、汇报一个按键值，至于什么 file\_operations、几种 I/O 模型，那是 Linux 的事情，为什么要我管？Linux 也是程序员写出来的，因此，程序员怎么想，它必然要怎么做。于是，这里就衍生出来了一个软件分层的想法，尽管 file\_operations、I/O 模型不可或缺，但是关于此部分的代码，全世界恐怕所有的输入设备都是一样的，为什么不提炼一个中间层出来，把这些事情搞定，也就是在底层编写驱动的时候，搞定具体的硬件操作呢？

将软件进行分层设计应该是软件工程最基本的一个思想，如果提炼一个 input 的核心层出来，把跟 Linux 接口以及整个一套 input 事件的汇报机制都在这里面实现，如图 12.2 所示，显然是非常好的。

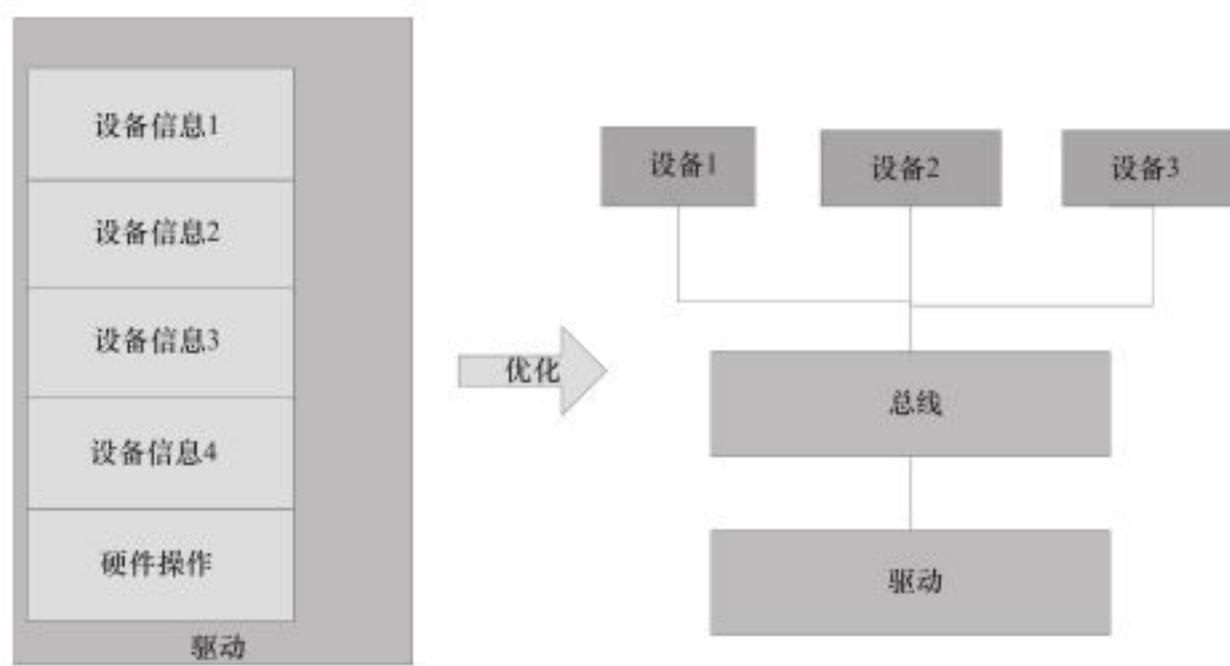


图 12.1 Linux 设备和驱动的分离

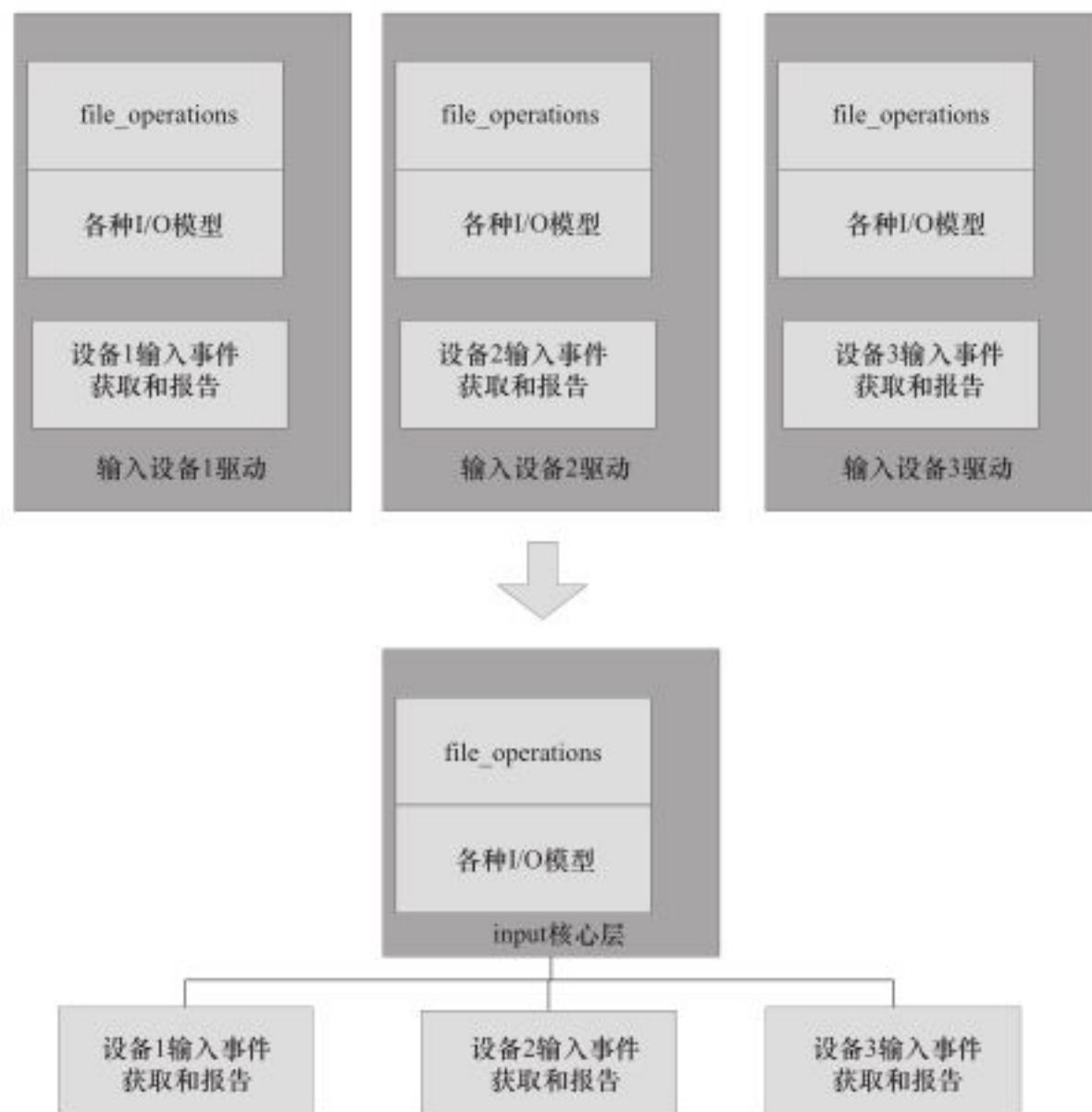


图 12.2 Linux 驱动的分层

在Linux设备驱动框架的设计中，除了有分层设计以外，还有分隔的思想。举一个简单的例子，假设我们要通过SPI总线访问某外设，假设CPU的名字叫XXX1，SPI外设叫YYY1。在访问YYY1外设的时候，要通过操作CPU XXX1上的SPI控制器的寄存器才能达到访问SPI外设YYY1的目的，最简单的代码逻辑是：

```
cpu_xxx1_spi_reg_write()
cpu_xxx1_spi_reg_read()
spi_client_yyy1_work1()
cpu_xxx1_spi_reg_write()
cpu_xxx1_spi_reg_read()
spi_client_yyy1_work2()
```

如果按照这种方式来设计驱动，结果对于任何一个SPI外设来讲，它的驱动代码都是与CPU相关的。也就是说，当代码用在CPU XXX1上的时候，它访问XXX1的SPI主机控制寄存器，当用在XXX2上的时候，它访问XXX2的SPI主机控制寄存器：

```
cpu_xxx2_spi_reg_write()
cpu_xxx2_spi_reg_read()
spi_client_yyy1_work1()
cpu_xxx2_spi_reg_write()
cpu_xxx2_spi_reg_read()
spi_client_yyy1_work2()
```

这显然是不被接受的，因为这意味着外设YYY1用在不同的CPU XXX1和XXX2上的时候需要不同的驱动。同时，如果CPU XXX1除了支持YYY1以外，还要支持外设YYY2、YYY3、YYY4等，这个XXX的代码就要重复出现在YYY1、YYY2、YYY3、YYY4的驱动里面：

```
cpu_xxx1_spi_reg_write()
cpu_xxx1_spi_reg_read()
spi_client_yyy2_work1()
cpu_xxx1_spi_reg_write()
cpu_xxx1_spi_reg_read()
spi_client_yyy2_work2()
...
```

按照这样的逻辑，如果要让N个不同的YYY在M个不同的CPU XXX上跑起来，需要 $M \times N$ 份代码。这是一种典型的强耦合，不符合软件工程“高内聚、低耦合”和“信息隐蔽”的基本原则。

这种软件架构是一种典型的网状耦合，网状耦合一般不太适合人类的思维逻辑，会把我们的思维搞乱。对于网状耦合的 $M:N$ ，我们一般要提炼出一个中间“1”，让M与“1”耦合，N也与这个“1”耦合，如图12.3所示。

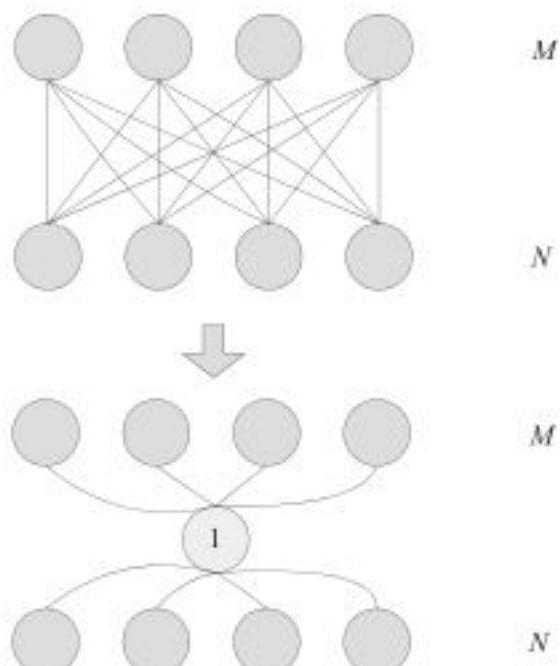


图 12.3 将 $M:N$ 耦合转化为 $M:1:N$ 耦合

那么，我们可以用如图 12.4 所示的思想对主机控制器驱动和外设驱动进行分离。这样的结果是，外设 YYY1、YYY2、YYY3、YYY4 的驱动与主机控制器 XXX1、XXX2、XXX3、XXX4 的驱动不相关，主机控制器驱动不关心外设，而外设驱动也不关心主机，外设只是访问核心层的通用 API 进行数据传输，主机和外设之间可以进行任意组合。

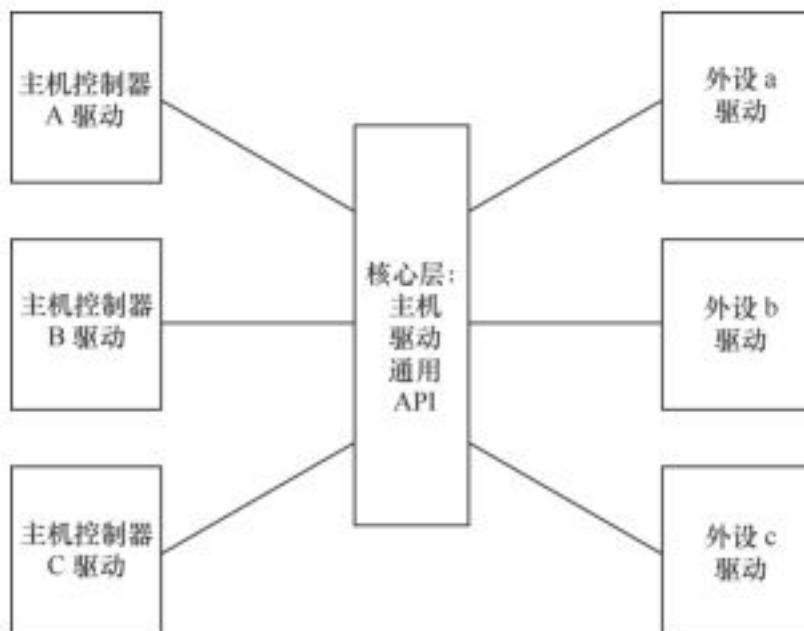


图 12.4 Linux 设备驱动的主机、外设驱动分离

如果我们不进行如图 12.4 所示的主机和外设分离，外设 YYY1、YYY2、YYY3 和主机 XXX1、XXX2、XXX3 进行组合的时候，需要 9 个不同的驱动。设想一共有  $m$  个主机控制器， $n$  个外设，分离的结果是需要  $m + n$  个驱动，不分离则需要  $m * n$  个驱动。因为， $m$  个主机控制器， $n$  个外设的驱动都可以被充分地复用了。

## 12.2 platform 设备驱动

### 12.2.1 platform 总线、设备与驱动

在 Linux 2.6 以后的设备驱动模型中，需关心总线、设备和驱动这 3 个实体，总线将设备和驱动绑定。在系统每注册一个设备的时候，会寻找与之匹配的驱动；相反的，在系统每注册一个驱动的时候，会寻找与之匹配的设备，而匹配由总线完成。

一个现实的 Linux 设备和驱动通常都需要挂接在一种总线上，对于本身依附于 PCI、USB、I<sup>2</sup>C、SPI 等的设备而言，这自然不是问题，但是在嵌入式系统里面，在 SoC 系统中集成的独立外设控制器、挂接在 SoC 内存空间的外设等却不依附于此类总线。基于这一背景，Linux 发明了一种虚拟的总线，称为 platform 总线，相应的设备称为 platform\_device，而驱动成为 platform\_driver。

注意：所谓的 platform\_device 并不是与字符设备、块设备和网络设备并列的概念，而是

Linux 系统提供的一种附加手段，例如，我们通常把在 SoC 内部集成的 I<sup>2</sup>C、RTC、LCD、看门狗等控制器都归纳为 platform\_device，而它们本身就是字符设备。platform\_device 结构体的定义如代码清单 12.1 所示。

代码清单 12.1 platform\_device 结构体

---

```

1 struct platform_device {
2     const char    *name;
3     int          id;
4     bool         id_auto;
5     struct devicedev;
6     u32          num_resources;
7     struct resource   *resource;
8
9     const struct platform_device_id      *id_entry;
10    char *driver_override; /* Driver name to force a match */
11
12    /* MFD cell pointer */
13    struct mfd_cell *mfd_cell;
14
15    /* arch specific additions */
16    struct pdev_archdata   archdata;
17 };

```

---

platform\_driver 这个结构体中包含 probe()、remove()、一个 device\_driver 实例、电源管理函数 suspend()、resume()，如代码清单 12.2 所示。

代码清单 12.2 platform\_driver 结构体

---

```

1 struct platform_driver {
2     int (*probe) (struct platform_device *);
3     int (*remove) (struct platform_device *);
4     void (*shutdown) (struct platform_device *);
5     int (*suspend) (struct platform_device *, pm_message_t state);
6     int (*resume) (struct platform_device *);
7     struct device_driver driver;
8     const struct platform_device_id *id_table;
9     bool prevent_deferred_probe;
10 };

```

---

直接填充 platform\_driver 的 suspend()、resume() 做电源管理回调的方法目前已经过时，较好的做法是实现 platform\_driver 的 device\_driver 中的 dev\_pm\_ops 结构体成员（后续的 Linux 电源管理章节会对此进行更细致的介绍），代码清单 12.3 给出了 device\_driver 的定义。

代码清单 12.3 device\_driver 结构体

---

```

1 struct device_driver {
2     const char           *name;
3     struct bus_type      *bus;

```

---

```

4
5     struct module          *owner;
6     const char           *mod_name; /* used for built-in modules */
7
8     bool suppress_bind_attrs; /* disables bind/unbind via sysfs */
9
10    const struct of_device_id      *of_match_table;
11    const struct acpi_device_id    *acpi_match_table;
12
13    int (*probe) (struct device *dev);
14    int (*remove) (struct device *dev);
15    void (*shutdown) (struct device *dev);
16    int (*suspend) (struct device *dev, pm_message_t state);
17    int (*resume) (struct device *dev);
18    const struct attribute_group **groups;
19
20    const struct dev_pm_ops *pm;
21
22    struct driver_private *p;
23 };

```

与 platform\_driver 地位对等的 i2c\_driver、spi\_driver、usb\_driver、pci\_driver 中都包含了 device\_driver 结构体实例成员。它其实描述了各种 xxx\_driver (xxx 是总线名) 在驱动意义上的一些共性。

系统为 platform 总线定义了一个 bus\_type 的实例 platform\_bus\_type，其定义位于 drivers/base/platform.c 下，如代码清单 12.4 所示。

代码清单 12.4 platform 总线的 bus\_type 实例 platform\_bus\_type

```

1 struct bus_type platform_bus_type = {
2     .name          = "platform",
3     .dev_groups    = platform_dev_groups,
4     .match         = platform_match,
5     .uevent        = platform_uevent,
6     .pm            = &platform_dev_pm_ops,
7 };

```

这里要重点关注其 match() 成员函数，正是此成员函数确定了 platform\_device 和 platform\_driver 之间是如何进行匹配，如代码清单 12.5 所示。

代码清单 12.5 platform\_bus\_type 的 match() 成员函数

```

1 static int platform_match(struct device *dev, struct device_driver *drv)
2 {
3     struct platform_device *pdev = to_platform_device(dev);
4     struct platform_driver *pdrv = to_platform_driver(drv);
5
6     /* Attempt an OF style match first */

```

```

7         if (of_driver_match_device(dev, drv))
8             return 1;
9
10        /* Then try ACPI style match */
11        if (acpi_driver_match_device(dev, drv))
12            return 1;
13
14        /* Then try to match against the id table */
15        if (pdev->id_table)
16            return platform_match_id(pdev->id_table, pdev) != NULL;
17
18        /* fall-back to driver name match */
19        return (strcmp(pdev->name, drv->name) == 0);
20    }

```

从代码清单 12.5 可以看出，匹配 platform\_device 和 platform\_driver 有 4 种可能性，一是基于设备树风格的匹配；二是基于 ACPI 风格的匹配；三是匹配 ID 表（即 platform\_device 设备名是否出现在 platform\_driver 的 ID 表内）；第四种是匹配 platform\_device 设备名和驱动的名字。

对于 Linux 2.6 ARM 平台而言，对 platform\_device 的定义通常在 BSP 的板文件中实现，在板文件中，将 platform\_device 归纳为一个数组，最终通过 platform\_add\_devices() 函数统一注册。platform\_add\_devices() 函数可以将平台设备添加到系统中，这个函数的原型为：

```
int platform_add_devices(struct platform_device **devs, int num);
```

该函数的第一个参数为平台设备数组的指针，第二个参数为平台设备的数量，它内部调用了 platform\_device\_register() 函数以注册单个的平台设备。

Linux 3.x 之后，ARM Linux 不太喜欢人们以编码的形式去填写 platform\_device 和注册，而倾向于根据设备树中的内容自动展开 platform\_device。

### 12.2.2 将 globalfifo 作为 platform 设备

现在我们将前面章节的 globalfifo 驱动挂接到 platform 总线上，这要完成两个工作。

- 1) 将 globalfifo 移植为 platform 驱动。
- 2) 在板文件中添加 globalfifo 这个 platform 设备。

为完成将 globalfifo 移植到 platform 驱动的工作，需要在原始的 globalfifo 字符设备驱动中套一层 platform\_driver 的外壳，如代码清单 12.6 所示。注意进行这一工作后，并没有改变 globalfifo 是字符设备的本质，只是将其挂接到了 platform 总线上。

代码清单 12.6 为 globalfifo 添加 platform\_driver

```

1 static int globalfifo_probe(struct platform_device *pdev)
2 {
3     int ret;

```

```
4     dev_t devno = MKDEV(globalfifo_major, 0);
5
6     if (globalfifo_major)
7         ret = register_chrdev_region(devno, 1, "globalfifo");
8     else {
9         ret = alloc_chrdev_region(&devno, 0, 1, "globalfifo");
10        globalfifo_major = MAJOR(devno);
11    }
12    if (ret < 0)
13        return ret;
14
15    globalfifo_devp = devm_kzalloc(&pdev->dev, sizeof(*globalfifo_devp),
16                                    GFP_KERNEL);
17    if (!globalfifo_devp) {
18        ret = -ENOMEM;
19        goto fail_malloc;
20    }
21
22    globalfifo_setup_cdev(globalfifo_devp, 0);
23
24    mutex_init(&globalfifo_devp->mutex);
25    init_waitqueue_head(&globalfifo_devp->r_wait);
26    init_waitqueue_head(&globalfifo_devp->w_wait);
27
28    return 0;
29
30 fail_malloc:
31     unregister_chrdev_region(devno, 1);
32     return ret;
33 }
34
35 static int globalfifo_remove(struct platform_device *pdev)
36 {
37     cdev_del(&globalfifo_devp->cdev);
38     unregister_chrdev_region(MKDEV(globalfifo_major, 0), 1);
39
40     return 0;
41 }
42
43 static struct platform_driver globalfifo_driver = {
44     .driver = {
45         .name = "globalfifo",
46         .owner = THIS_MODULE,
47     },
48     .probe = globalfifo_probe,
49     .remove = globalfifo_remove,
50 };
51 module_platform_driver(globalfifo_driver);
```

在代码清单 12.6 中, `module_platform_driver()` 宏所定义的模块加载和卸载函数仅仅通过 `platform_driver_register()`、`platform_driver_unregister()` 函数进行 `platform_driver` 的注册与注销, 而原先注册和注销字符设备的工作已经被移交到 `platform_driver` 的 `probe()` 和 `remove()` 成员函数中。

代码清单 12.6 未列出的部分与原始的 `globalfifo` 驱动相同, 都是实现作为字符设备驱动核心的 `file_operations` 的成员函数。注册完 `globalfifo` 对应的 `platform_driver` 后, 我们会发现 `/sys/bus/platform/drivers` 目录下多出了一个名字叫 `globalfifo` 的子目录。

为了完成在板文件中添加 `globalfifo` 这个 `platform` 设备的工作, 需要在板文件 `arch/arm/mach-<soc 名>/mach-<板名>.c` 中添加相应的代码, 如代码清单 12.7 所示。

代码清单 12.7 与 `globalfifo` 对应的 `platform_device`

---

```

1 static struct platform_device globalfifo_device = {
2     .name      = "globalfifo",
3     .id        = -1,
4 };

```

---

并最终通过类似于 `platform_add_devices()` 的函数把这个 `platform_device` 注册进系统。如果一切顺利, 我们会在 `/sys/devices/platform` 目录下看到一个名字叫 `globalfifo` 的子目录, `/sys/devices/platform/globalfifo` 中会有一个 `driver` 文件, 它是指向 `/sys/bus/platform/drivers/globalfifo` 的符号链接, 这证明驱动和设备匹配上了。

### 12.2.3 platform 设备资源和数据

留意一下代码清单 12.1 中 `platform_device` 结构体定义的第 6 ~ 7 行, 它们描述了 `platform_device` 的资源, 资源本身由 `resource` 结构体描述, 其定义如代码清单 12.8 所示。

代码清单 12.8 `resource` 结构体定义

---

```

1 struct resource {
2     resource_size_t start;
3     resource_size_t end;
4     const char *name;
5     unsigned long flags;
6     struct resource *parent, *sibling, *child;
7 };

```

---

我们通常关心 `start`、`end` 和 `flags` 这 3 个字段, 它们分别标明了资源的开始值、结束值和类型, `flags` 可以为 `IORESOURCE_IO`、`IORESOURCE_MEM`、`IORESOURCE_IRQ`、`IORESOURCE_DMA` 等。`start`、`end` 的含义会随着 `flags` 而变更, 如当 `flags` 为 `IORESOURCE_MEM` 时, `start`、`end` 分别表示该 `platform_device` 占据的内存的开始地址和结束地址; 当 `flags` 为 `IORESOURCE_IRQ` 时, `start`、`end` 分别表示该 `platform_device` 使用的中断号的开始值和

结束值，如果只使用了1个中断号，开始和结束值相同。对于同种类型的资源而言，可以有多份，比如说某设备占据了两个内存区域，则可以定义两个 IORESOURCE\_MEM 资源。

对 resource 的定义也通常在 BSP 的板文件中进行，而在具体的设备驱动中通过 platform\_get\_resource() 这样的 API 来获取，此 API 的原型为：

```
struct resource *platform_get_resource(struct platform_device *, unsigned int,
                                       unsigned int);
```

例如在 arch/arm/mach-at91/board-sam9261ek.c 板文件中为 DM9000 网卡定义了如下 resource：

```
static struct resource dm9000_resource[] = {
    [0] = {
        .start  = AT91_CHIPSELECT_2,
        .end    = AT91_CHIPSELECT_2 + 3,
        .flags  = IORESOURCE_MEM
    },
    [1] = {
        .start  = AT91_CHIPSELECT_2 + 0x44,
        .end    = AT91_CHIPSELECT_2 + 0xFF,
        .flags  = IORESOURCE_MEM
    },
    [2] = {
        .flags  = IORESOURCE_IRQ
            | IORESOURCE_IRQ_LOWEDGE | IORESOURCE_IRQ_HIGHEDGE,
    }
};
```

在 DM9000 网卡的驱动中则是通过如下办法拿到这 3 份资源：

```
db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);
db->irq_res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
```

对于 IRQ 而言，platform\_get\_resource() 还有一个进行了封装的变体 platform\_get\_irq()，其原型为：

```
int platform_get_irq(struct platform_device *dev, unsigned int num);
```

它实际上调用了“platform\_get\_resource(dev, IORESOURCE\_IRQ, num);”。

设备除了可以在 BSP 中定义资源以外，还可以附加一些数据信息，因为对设备的硬件描述除了中断、内存等标准资源以外，可能还会有一些配置信息，而这些配置信息也依赖于板，不适宜直接放置在设备驱动上。因此，platform 也提供了 platform\_data 的支持，platform\_data 的形式是由每个驱动自定义的，如对于 DM9000 网卡而言，platform\_data 为一个 dm9000\_plat\_data 结构体，完成定义后，就可以将 MAC 地址、总线宽度、板上有无 EEPROM 信息等放入 platform\_data 中，如代码清单 12.9 所示。

代码清单 12.9 platform\_data 的使用

---

```

1 static struct dm9000_plat_data dm9000_platdata = {
2     .flags          = DM9000_PLATF_16BITONLY | DM9000_PLATF_NO_EEPROM,
3 };
4
5 static struct platform_device dm9000_device = {
6     .name           = "dm9000",
7     .id             = 0,
8     .num_resources  = ARRAY_SIZE(dm9000_resource),
9     .resource       = dm9000_resource,
10    .dev            = {
11        .platform_data = &dm9000_platdata,
12    }
13 };

```

---

而在 DM9000 网卡的驱动 drivers/net/ethernet/davicom/dm9000.c 的 probe() 中，通过如下方式就拿到了 platform\_data：

```
struct dm9000_plat_data *pdata = dev_get_platdata(&pdev->dev);
```

其中，pdev 为 platform\_device 的指针。

由以上分析可知，在设备驱动中引入 platform 的概念至少有如下好处。

1) 使得设备被挂接在一个总线上，符合 Linux 2.6 以后内核的设备模型。其结果是使配套的 sysfs 节点、设备电源管理都成为可能。

2) 隔离 BSP 和驱动。在 BSP 中定义 platform 设备和设备使用的资源、设备的具体配置信息，而在驱动中，只需要通过通用 API 去获取资源和数据，做到了板相关代码和驱动代码的分离，使得驱动具有更好的可扩展性和跨平台性。

3) 让一个驱动支持多个设备实例。譬如 DM9000 的驱动只有一份，但是我们可以在板级添加多份 DM9000 的 platform\_device，它们都可以与唯一的驱动匹配。

在 Linux 3.x 之后的内核中，DM9000 驱动实际上已经可以通过设备树的方法被枚举，可以参见补丁 net: dm9000: Allow instantiation using device tree (内核 commit 的 ID 是 0b8bf1ba)。

```

index a2408c8..dd243a1 100644
--- a/drivers/net/ethernet/davicom/dm9000.c
+++ b/drivers/net/ethernet/davicom/dm9000.c
@@ -29,6 +29,8 @@ 
#include <linux/spinlock.h>
#include <linux/crc32.h>
#include <linux/mii.h>
+#include <linux/of.h>
+#include <linux/of_net.h>
#include <linux/ethtool.h>
#include <linux/dm9000.h>
#include <linux/delay.h>
@@ -1351,6 +1353,31 @@ static const struct net_device_ops dm9000_netdev_ops = {
#endif
};

```

```

+static struct dm9000_plat_data *dm9000_parse_dt(struct device *dev)
+{
+ ...
+}
+
/*
 * Search DM9000 board, allocate space and register it
 */
@@ -1366,6 +1393,12 @@ dm9000_probe(struct platform_device *pdev)
int i;
u32 id_val;
+ if (!pdata) {
+     pdata = dm9000_parse_dt(&pdev->dev);
+     if (IS_ERR(pdata))
+         return PTR_ERR(pdata);
+ }
+
/* Init network device */
ndev = alloc_etherdev(sizeof(struct board_info));
if (!ndev)
@@ -1676,11 +1709,20 @@ dm9000_drv_remove(struct platform_device *pdev)
return 0;
}
+#ifdef CONFIG_OF
+static const struct of_device_id dm9000_of_matches[] = {
+    { .compatible = "davicom,dm9000", },
+    { /* sentinel */ }
+};
+MODULE_DEVICE_TABLE(of, dm9000_of_matches);
+#endif
+
static struct platform_driver dm9000_driver = {
    .driver = {
        .name = "dm9000",
        .owner = THIS_MODULE,
        .pm = &dm9000_drv_pm_ops,
+        .of_match_table = of_match_ptr(dm9000_of_matches),
    },
    .probe = dm9000_probe,
    .remove = dm9000_drv_remove,

```

改为设备树后，在板上添加DM9000网卡的动作就变成了简单地修改dts文件，如arch/arm/boot/dts/s3c6410-mini6410.dts中就有这样的代码：

```

srom-cs1@18000000 {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    reg = <0x18000000 0x8000000>;
    ranges;

    ethernet@18000000 {

```

```

        compatible = "davicom,dm9000";
        reg = <0x18000000 0x2 0x18000004 0x2>;
        interrupt-parent = <&gpn>;
        interrupts = <7 IRQ_TYPE_LEVEL_HIGH>;
        davicom,no-eeprom;
    };
};


```

关于设备树情况下驱动与设备的匹配，以及驱动如何获取平台属性的更详细细节，将在后续章节介绍。

## 12.3 设备驱动的分层思想

### 12.3.1 设备驱动核心层和例化

在 12.1 节，我们已经从感性上认识了 Linux 驱动软件分层的意义。其实，在分层设计的时候，Linux 内核大量使用了面向对象的设计思想。

在面向对象的程序设计中，可以为某一类相似的事物定义一个基类，而具体的事物可以继承这个基类中的函数。如果对于继承的这个事物而言，某成员函数的实现与基类一致，那它就可以直接继承基类的函数；相反，它也可以重写（Overriding），对父类的函数进行重新定义。若子类中的方法与父类中的某方法具有相同的方法名、返回类型和参数表，则新方法将覆盖原有的方法。这种面向对象的“多态”设计思想极大地提高了代码的可重用能力，是对现实世界中事物之间关系的一种良好呈现。

Linux 内核完全是由 C 语言和汇编语言写成，但是却频繁地用到了面向对象的设计思想。在设备驱动方面，往往为同类的设备设计了一个框架，而框架中的核心层则实现了该设备通用的一些功能。同样的，如果具体的设备不想使用核心层的函数，也可以重写。举个例子：

```

return_type core_funca(xxx_device * bottom_dev, param1_type param1, param1_type param2)
{
    if (bottom_dev->funca)
        return bottom_dev->funca(param1, param2);
    /* 核心层通用的 funca 代码 */
    ...
}


```

在上述 core\_funca 的实现中，会检查底层设备是否重写了 funca()，如果重写了，就调用底层的代码，否则，直接使用通用层的。这样做的好处是，核心层的代码可以处理绝大多数与该类设备的 funca() 对应的功能，只有少数特殊设备需要重新实现 funca()。

再看一个例子：

```

return_type core_funca(xxx_device * bottom_dev, param1_type param1, param1_type param2)
{
    /* 通用的步骤代码 A */
}


```

```

typea_dev_commonA();
...
/* 底层操作 ops1 */
bottom_dev->funca_ops1();

/* 通用的步骤代码 B */
typea_dev_commonB();
...

/* 底层操作 ops2 */
bottom_dev->funca_ops2();

/* 通用的步骤代码 C */
typea_dev_commonB();
...

/** 底层操作 ops3 */
bottom_dev->funca_ops3();
}

```

上述代码假定为了实现 funca()，对于同类设备而言，操作流程一致，都要经过“通用代码 A、底层 ops1、通用代码 B、底层 ops2、通用代码 C、底层 ops3”这几步，分层设计带来的明显好处是，对于通用代码 A、B、C，具体的底层驱动不需要再实现，而仅仅只要关心其底层的操作 ops1、ops2、ops3 则可。

图 12.5 明确反映了设备驱动的核心层与具体设备驱动的关系，实际上，这种分层可能只有两层（见图 12.5a），也可能是多层的（图 12.5b）。

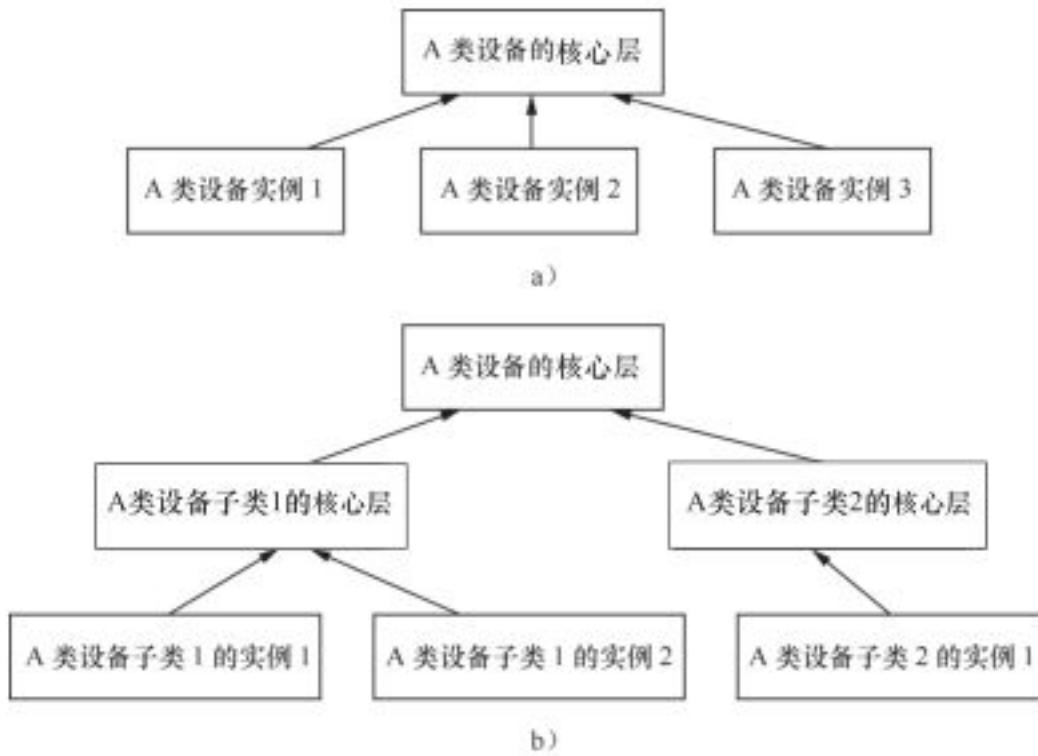


图 12.5 Linux 设备驱动的分层

这样的分层化设计在 Linux 的 input、RTC、MTD、I<sup>2</sup>C、SPI、tty、USB 等诸多类型设备驱动中屡见不鲜。下面的几小节以 input、RTC、Framebuffer 等为例先进行一番讲解，当然，后续的章节会对与几个大的设备类型对应的驱动层次进行更详细的分析。

### 12.3.2 输入设备驱动

输入设备（如按键、键盘、触摸屏、鼠标等）是典型的字符设备，其一般的工作机理是底层在按键、触摸等动作发送时产生一个中断（或驱动通过 Timer 定时查询），然后 CPU 通过 SPI、I<sup>2</sup>C 或外部存储器总线读取键值、坐标等数据，并将它们放入一个缓冲区，字符设备驱动管理该缓冲区，而驱动的 read() 接口让用户可以读取键值、坐标等数据。

显然，在这些工作中，只是中断、读键值 / 坐标值是与设备相关的，而输入事件的缓冲区管理以及字符设备驱动的 file\_operations 接口则对输入设备是通用的。基于此，内核设计了输入子系统，由核心层处理公共的工作。Linux 内核输入子系统的框架如图 12.6 所示。

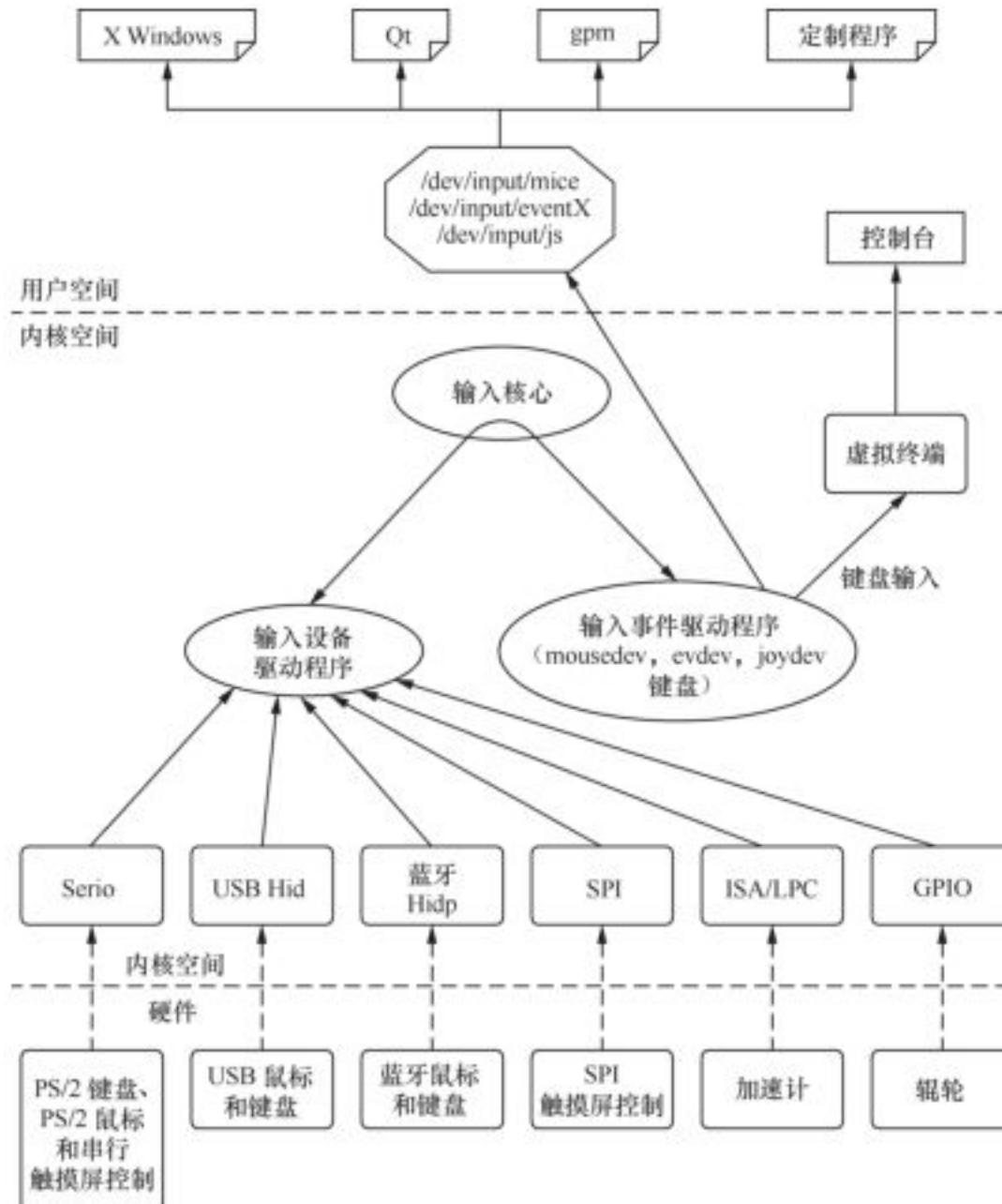


图 12.6 Linux 内核输入子系统的框架

输入核心提供了底层输入设备驱动程序所需的 API，如分配 / 释放一个输入设备：

```
struct input_dev *input_allocate_device(void);
void input_free_device(struct input_dev *dev);
```

`input_allocate_device()` 返回的是 1 个 `input_dev` 的结构体，此结构体用于表征 1 个输入设备。

注册 / 注销输入设备用的接口如下：

```
int __must_check input_register_device(struct input_dev *dev);
void input_unregister_device(struct input_dev *dev);
```

报告输入事件用的接口如下：

```
/* 报告指定 type, code 的输入事件 */
void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);
/* 报告键值 */
void input_report_key(struct input_dev *dev, unsigned int code, int value);
/* 报告相对坐标 */
void input_report_rel(struct input_dev *dev, unsigned int code, int value);
/* 报告绝对坐标 */
void input_report_abs(struct input_dev *dev, unsigned int code, int value);
/* 报告同步事件 */
void input_sync(struct input_dev *dev);
```

而对于所有的输入事件，内核都用统一的数据结构来描述，这个数据结构是 `input_event`，如代码清单 12.10 所示。

代码清单 12.10 `input_event` 结构体

---

```
1 struct input_event {
2     struct timeval time;
3     __u16 type;
4     __u16 code;
5     __s32 value;
6 };
```

---

`drivers/input/keyboard/gpio_keys.c` 基于 `input` 架构实现了一个通用的 GPIO 按键驱动。该驱动是基于 `platform_driver` 架构的，名为“`gpio-keys`”。它将与硬件相关的信息（如使用的 GPIO 号，按下和抬起时的电平等）屏蔽在板文件 `platform_device` 的 `platform_data` 中，因此该驱动可应用于各个处理器，具有良好的跨平台性。代码清单 12.11 列出了该驱动的 `probe()` 函数。

代码清单 12.11 GPIO 按键驱动的 `probe()` 函数

---

```
1 static int gpio_keys_probe(struct platform_device *pdev)
2 {
3     struct device *dev = &pdev->dev;
4     const struct gpio_keys_platform_data *pdata = dev_get_platdata(dev);
5     struct gpio_keys_drvdata *ddata;
6     struct input_dev *input;
7     size_t size;
```

```
8 int i, error;
9 int wakeup = 0;
10
11 if (!pdata) {
12     pdata = gpio_keys_get_devtree_pdata(dev);
13     if (IS_ERR(pdata))
14         return PTR_ERR(pdata);
15 }
16
17 size = sizeof(struct gpio_keys_drvdata) +
18         pdata->nbuttons * sizeof(struct gpio_button_data);
19 ddata = devm_kzalloc(dev, size, GFP_KERNEL);
20 if (!ddata) {
21     dev_err(dev, "failed to allocate state\n");
22     return -ENOMEM;
23 }
24
25 input = devm_input_allocate_device(dev);
26 if (!input) {
27     dev_err(dev, "failed to allocate input device\n");
28     return -ENOMEM;
29 }
30
31 ddata->pdata = pdata;
32 ddata->input = input;
33 mutex_init(&ddata->disable_lock);
34
35 platform_set_drvdata(pdev, ddata);
36 input_set_drvdata(input, ddata);
37
38 input->name = pdata->name ? : pdev->name;
39 input->phys = "gpio-keys/input0";
40 input->dev.parent = &pdev->dev;
41 input->open = gpio_keys_open;
42 input->close = gpio_keys_close;
43
44 input->id.bustype = BUS_HOST;
45 input->id.vendor = 0x0001;
46 input->id.product = 0x0001;
47 input->id.version = 0x0100;
48
49 /* Enable auto repeat feature of Linux input subsystem */
50 if (pdata->rep)
51     __set_bit(EV_REP, input->evbit);
52
53 for (i = 0; i < pdata->nbuttons; i++) {
54     const struct gpio_keys_button *button = &pdata->buttons[i];
55     struct gpio_button_data *bdata = &ddata->data[i];
56
57     error = gpio_keys_setup_key(pdev, input, bdata, button);
58     if (error)
```

```

59         return error;
60
61     if (button->wakeup)
62         wakeup = 1;
63 }
64
65 error = sysfs_create_group(&pdev->dev.kobj, &gpio_keys_attr_group);
66 ...
67 error = input_register_device(input);
68 ...
69 }

```

上述代码的第 25 行分配了 1 个输入设备，第 31 ~ 47 行初始化了该 `input_dev` 的一些属性，第 58 行注册了这个输入设备。第 53 ~ 63 行则初始化了所用到的 GPIO，第 67 行完成了这个输入设备的注册。

在注册输入设备后，底层输入设备驱动的核心工作只剩下在按键、触摸等人为动作发生时报告事件。代码清单 12.12 列出了 GPIO 按键中断发生时的事件报告代码。

代码清单 12.12 GPIO 按键中断发生时的事件报告

```

1 static irqreturn_t gpio_keys_irq_isr(int irq, void *dev_id)
2 {
3     struct gpio_button_data *bdata = dev_id;
4     const struct gpio_keys_button *button = bdata->button;
5     struct input_dev *input = bdata->input;
6     unsigned long flags;
7
8     BUG_ON(irq != bdata->irq);
9
10    spin_lock_irqsave(&bdata->lock, flags);
11
12    if (!bdata->key_pressed) {
13        if (bdata->button->wakeup)
14            pm_wakeup_event(bdata->input->dev.parent, 0);
15
16        input_event(input, EV_KEY, button->code, 1);
17        input_sync(input);
18
19        if (!bdata->timer_debounce) {
20            input_event(input, EV_KEY, button->code, 0);
21            input_sync(input);
22            goto out;
23        }
24
25        bdata->key_pressed = true;
26    }
27
28    if (bdata->timer_debounce)
29        mod_timer(&bdata->timer,

```

```

30         jiffies + msecs_to_jiffies(bdata->timer_debounce));
31     out:
32     spin_unlock_irqrestore(&bdata->lock, flags);
33     return IRQ_HANDLED;
34 }

```

GPIO 按键驱动通过 `input_event()`、`input_sync()` 这样的函数来汇报按键事件以及同步事件。从底层的 GPIO 按键驱动可以看出，该驱动中没有任何 `file_operations` 的动作，也没有各种 I/O 模型，注册进入系统也用的是 `input_register_device()` 这样的与 `input` 相关的 API。这是由于与 Linux VFS 接口的这一部分代码全部都在 `drivers/input/evdev.c` 中实现了，代码清单 12.13 摘取了部分关键代码。

代码清单 12.13 `input` 核心层的 `file_operations` 和 `read()` 函数

```

1 static ssize_t evdev_read(struct file *file, char __user *buffer,
2                         size_t count, loff_t *ppos)
3 {
4     struct evdev_client *client = file->private_data;
5     struct evdev *evdev = client->evdev;
6     struct input_event event;
7     size_t read = 0;
8     int error;
9
10    if (count != 0 && count < input_event_size())
11        return -EINVAL;
12
13    for (;;) {
14        if (!evdev->exist || client->revoked)
15            return -ENODEV;
16
17        if (client->packet_head == client->tail &&
18            (file->f_flags & O_NONBLOCK))
19            return -EAGAIN;
20
21        /*
22         * count == 0 is special - no IO is done but we check
23         * for error conditions (see above).
24         */
25        if (count == 0)
26            break;
27
28        while (read + input_event_size() <= count &&
29               evdev_fetch_next_event(client, &event)) {
30
31            if (input_event_to_user(buffer + read, &event))
32                return -EFAULT;
33
34            read += input_event_size();
35        }
36    }

```

```

37     if (read)
38         break;
39
40     if (!(file->f_flags & O_NONBLOCK)) {
41         error = wait_event_interruptible(evdev->wait,
42                                         client->packet_head != client->tail ||
43                                         !evdev->exist || client->revoked);
44         if (error)
45             return error;
46     }
47 }
48
49 return read;
50 }
51
52 static const struct file_operations evdev_fops = {
53     .owner          = THIS_MODULE,
54     .read           = evdev_read,
55     .write          = evdev_write,
56     .poll           = evdev_poll,
57     .open            = evdev_open,
58     .release         = evdev_release,
59     .unlocked_ioctl= evdev_ioctl,
60 #ifdef CONFIG_COMPAT
61     .compat_ioctl   = evdev_ioctl_compat,
62 #endif
63     .fasync          = evdev_fasync,
64     .flush           = evdev_flush,
65     .llseek          = no_llseek,
66 };

```

---

上述代码中的 17 ~ 19 行在检查出是非阻塞访问后，立即返回 EAGAIN 错误，而第 29 行和第 41 ~ 43 行的代码则处理了阻塞的睡眠情况。回过头来想，其实 gpio\_keys 驱动里面调用的 input\_event()、input\_sync() 有间接唤醒这个等待队列 evdev->wait 的功能，只不过这些代码都隐藏在其内部实现里了。

### 12.3.3 RTC 设备驱动

RTC（实时钟）借助电池供电，在系统掉电的情况下依然可以正常计时。它通常还具有产生周期性中断以及闹钟（Alarm）中断的能力，是一种典型的字符设备。作为一种字符设备驱动，RTC 需要有 file\_operations 中接口函数的实现，如 open()、release()、read()、poll()、ioctl() 等，而典型的 IOCTL 包括 RTC\_SET\_TIME、RTC\_ALM\_READ、RTC\_ALM\_SET、RTC\_IRQP\_SET、RTC\_IRQP\_READ 等，这些对于所有的 RTC 是通用的，只有底层的具体实现是与设备相关的。

因此，drivers/rtc/rtc-dev.c 实现了 RTC 驱动通用的字符设备驱动层，它实现了 file\_operations 的成员函数以及一些通用的关于 RTC 的控制代码，并向底层导出 rtc\_device\_register()、rtc\_device\_unregister() 以注册和注销 RTC；导出 rtc\_class\_ops 结构体以描述底层的

RTC 硬件操作。这个 RTC 通用层实现的结果是，底层的 RTC 驱动不再需要关心 RTC 作为字符设备驱动的具体实现，也无需关心一些通用的 RTC 控制逻辑，图 12.7 表明了这种关系。

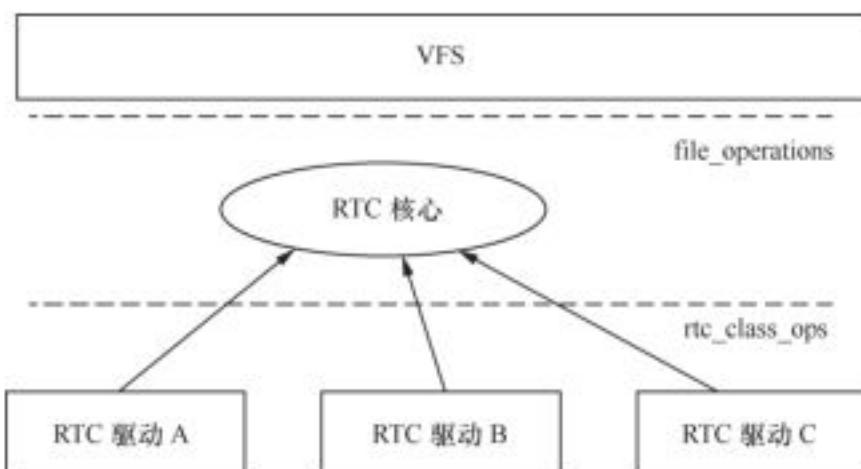


图 12.7 Linux RTC 设备驱动的分层

`drivers/rtc/rtc-s3c.c` 实现了 S3C6410 的 RTC 驱动，其注册 RTC 以及绑定 `rtc_class_ops` 的代码如代码清单 12.14 所示。

代码清单 12.14 S3C6410 RTC 驱动的 `rtc_class_ops` 实例与 RTC 注册

---

```

1 static const struct rtc_class_ops s3c_rtcops = {
2     .read_time      = s3c_rtc_gettime,
3     .set_time       = s3c_rtc_settime,
4     .read_alarm     = s3c_rtc_getalarm,
5     .set_alarm      = s3c_rtc_setalarm,
6     .proc           = s3c_rtc_proc,
7     .alarm_irq_enable = s3c_rtc_setaie,
8 };
9
10 static int s3c_rtc_probe(struct platform_device *pdev)
11 {
12     ...
13     rtc = devm_rtc_device_register(&pdev->dev, "s3c", &s3c_rtcops,
14                                     THIS_MODULE);
15     ...
16 }

```

---

`drivers/rtc/rtc-dev.c` 以及其调用的 `drivers/rtc/interface.c` 等 RTC 核心层相当于把 `file_operations` 中的 `open()`、`release()`、读取和设置时间等都间接“转发”给了底层的实例，代码清单 12.15 摘取了部分 RTC 核心层调用具体底层驱动 `callback` 的过程。

代码清单 12.15 RTC 核心层“转发”到底层 RTC 驱动 `callback`

---

```

1 static int rtc_dev_open(struct inode *inode, struct file *file)
2 {
3     ...
4     err = ops->open ? ops->open(rtc->dev.parent) : 0;

```

---

```
5 ...
6 }
7
8 static int __rtc_read_time(struct rtc_device *rtc, struct rtc_time *tm)
9 {
10    int err;
11    if (!rtc->ops)
12        err = -ENODEV;
13    else if (!rtc->ops->read_time)
14        err = -EINVAL;
15 ...
16    return err;
17 }
18
19 int rtc_read_time(struct rtc_device *rtc, struct rtc_time *tm)
20 {
21    int err;
22
23    err = mutex_lock_interruptible(&rtc->ops_lock);
24    if (err)
25        return err;
26
27    err = __rtc_read_time(rtc, tm);
28    mutex_unlock(&rtc->ops_lock);
29    return err;
30 }
31
32 int rtc_set_time(struct rtc_device *rtc, struct rtc_time *tm)
33 {
34 ...
35
36    if (!rtc->ops)
37        err = -ENODEV;
38    else if (rtc->ops->set_time)
39        err = rtc->ops->set_time(rtc->dev.parent, tm);
40 ...
41    return err;
42 }
43
44 static long rtc_dev_ioctl(struct file *file,
45     unsigned int cmd, unsigned long arg)
46 {
47 ...
48
49 case RTC_RD_TIME:
50     mutex_unlock(&rtc->ops_lock);
51
52     err = rtc_read_time(rtc, &tm);
53     if (err < 0)
54         return err;
55 }
```

```

56     if (copy_to_user(uarg, &tm, sizeof(tm)))
57         err = -EFAULT;
58     return err;
59
60     case RTC_SET_TIME:
61         mutex_unlock(&rtc->ops_lock);
62
63         if (copy_from_user(&tm, uarg, sizeof(tm)))
64             return -EFAULT;
65
66         return rtc_set_time(rtc, &tm);
67     ...
68 }

```

### 12.3.4 Framebuffer 设备驱动

Framebuffer（帧缓冲）是 Linux 系统为显示设备提供的一个接口，它将显示缓冲区抽象，屏蔽图像硬件的底层差异，允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。对于帧缓冲设备而言，只要在显示缓冲区中与显示点对应的区域内写入颜色值，对应的颜色会自动在屏幕上显示。

图 12.8 所示为 Linux 帧缓冲设备驱动的主要结构，帧缓冲设备提供给用户空间的 file\_operations 结构体由 drivers/video/fbdev/core/fbmem.c 中的 file\_operations 提供，而特定帧缓冲设备 fb\_info 结构体的注册、注销以及其中成员的维护，尤其是 fb\_ops 中成员函数的实现则由对应的 xxxfb.c 文件实现，fb\_ops 中的成员函数最终会操作 LCD 控制器。

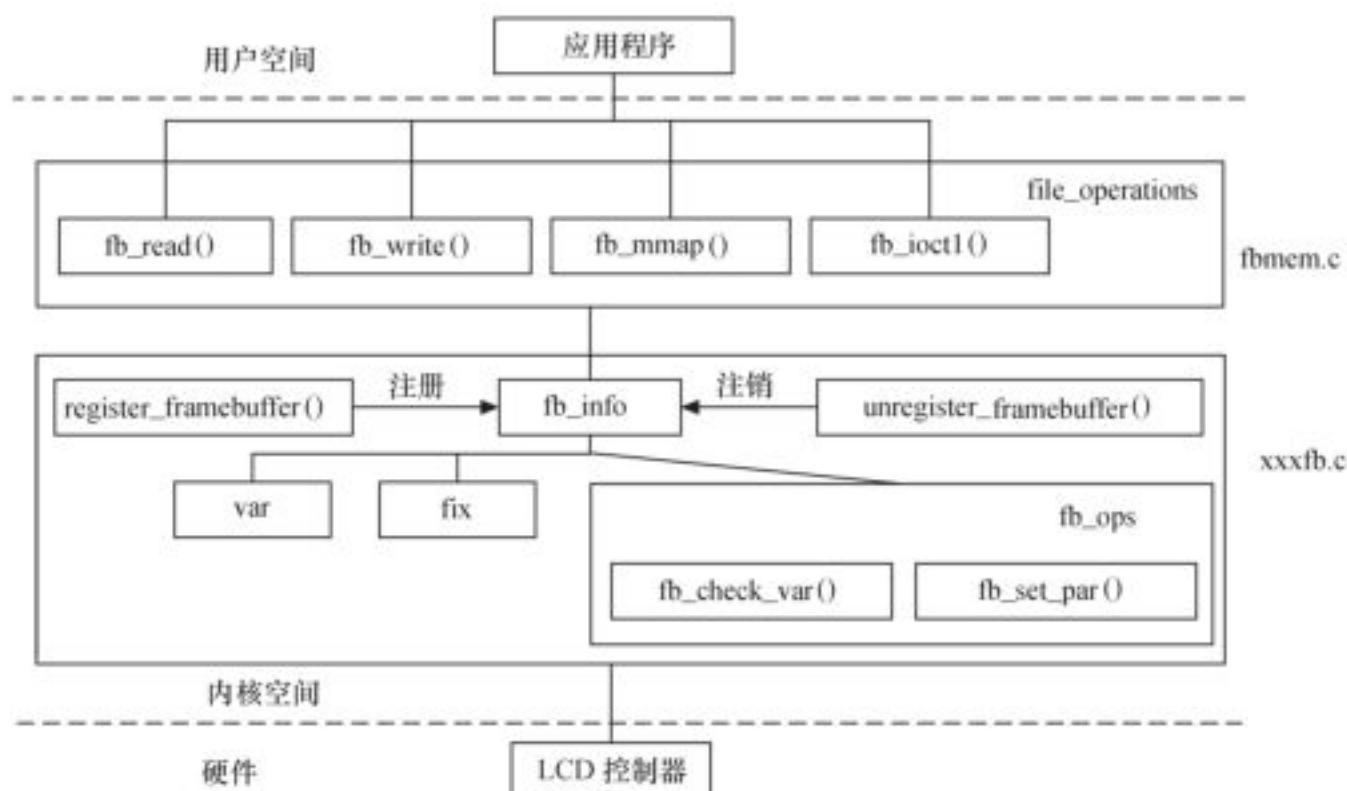


图 12.8 Linux 帧缓冲设备驱动的程序结构

多数显存的操作方法都是规范的，可以按照像素点格式的要求顺序写帧缓冲区。但是有少量LCD的显存写法可能比较特殊，这时候，在核心层 drivers/video/fbdev/core/fbmem.c 实现的 fb\_write() 中，实际上可以给底层提供一个重写自己的机会，如代码清单 12.16 所示。

代码清单 12.16 LCD 的 framebuffer write() 函数

```

1 static ssize_t
2 fb_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
3 {
4     unsigned long p = *ppos;
5     struct fb_info *info = file_fb_info(file);
6     u8 *buffer, *src;
7     u8 __iomem *dst;
8     int c, cnt = 0, err = 0;
9     unsigned long total_size;
10
11    if (!info || !info->screen_base)
12        return -ENODEV;
13
14    if (info->state != FBINFO_STATE_RUNNING)
15        return -EPERM;
16
17    if (info->fbops->fb_write)
18        return info->fbops->fb_write(info, buf, count, ppos);
19
20    total_size = info->screen_size;
21
22    if (total_size == 0)
23        total_size = info->fix.smem_len;
24
25    if (p > total_size)
26        return -EFBIG;
27
28    if (count > total_size) {
29        err = -EFBIG;
30        count = total_size;
31    }
32
33    if (count + p > total_size) {
34        if (!err)
35            err = -ENOSPC;
36
37        count = total_size - p;
38    }
39
40    buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count,
41                     GFP_KERNEL);
42    if (!buffer)
43        return -ENOMEM;
44

```

```

45     dst = (u8 __iomem *) (info->screen_base + p);
46
47     if (info->fbops->fb_sync)
48         info->fbops->fb_sync(info);
49
50     while (count) {
51         c = (count > PAGE_SIZE) ? PAGE_SIZE : count;
52         src = buffer;
53
54         if (copy_from_user(src, buf, c)) {
55             err = -EFAULT;
56             break;
57         }
58
59         fb_memcpy_tofb(dst, src, c);
60         dst += c;
61         src += c;
62         *ppos += c;
63         buf += c;
64         cnt += c;
65         count -= c;
66     }
67
68     kfree(buffer);
69
70     return (cnt) ? cnt : err;
71 }

```

第 17 ~ 18 行是一个检查底层 LCD 有没有实现自己特殊显存写法的代码，如果有，直接调底层的；如果没有，用中间层标准的显存写法就搞定了底层的那个不特殊的 LCD。

### 12.3.5 终端设备驱动

在 Linux 系统中，终端是一种字符型设备，它有多种类型，通常使用 tty (Teletype) 来简称各种类型的终端设备。对于嵌入式系统而言，最普遍采用的是 UART ( Universal Asynchronous Receiver/Transmitter) 串行端口，日常生活中简称串口。

Linux 内核中 tty 的层次结构如图 12.9 所示，它包含 tty 核心 tty\_io.c、tty 线路规程 n\_tty.c (实现 N\_TTY 线路规程) 和 tty 驱动实例 xxx\_tty.c，tty 线路规程的工作是以特殊的方式格式化从一个用户或者硬件收到的数据，这种格式化常常采用一个协议转换的形式。

tty\_io.c 本身是一个标准的字符设备驱动，它对上有字符设备的职责，实现 file\_operations 成员函数。但是 tty 核心层对下又定义了 tty\_driver 的架构，这样 tty 设备驱动的主体工作就变成了填充 tty\_driver 结构体中的成员，实现其中的 tty\_operations 的成员函数，而不再是去实现 file\_operations 这一级的工作。tty\_driver 结构体和 tty\_operations 的定义分别如代码清单 12.17 和 12.18 所示。

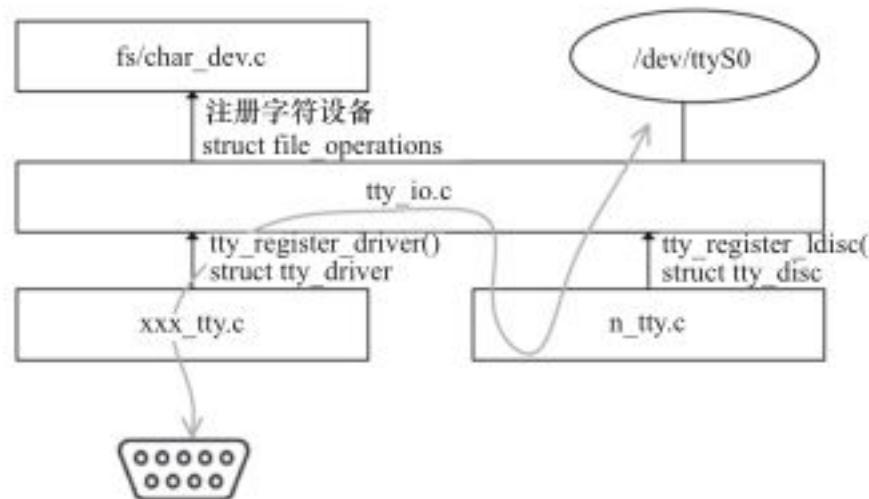


图 12.9 Linux 内核中 tty 的层次结构

## 代码清单 12.17 tty\_driver 结构体

```

1 struct tty_driver {
2     int      magic;           /* magic number for this structure */
3     struct kref kref;         /* Reference management */
4     struct cdev *cdevs;
5     struct module *owner;
6     const char *driver_name;
7     const char *name;
8     int      name_base;       /* offset of printed name */
9     int      major;           /* major device number */
10    int     minor_start;      /* start of minor device number */
11    unsigned int num;          /* number of devices allocated */
12    short   type;             /* type of tty driver */
13    short   subtype;          /* subtype of tty driver */
14    struct ktermios init_termios; /* Initial termios */
15    unsigned long flags;       /* tty driver flags */
16    struct proc_dir_entry *proc_entry; /* /proc fs entry */
17    struct tty_driver *other;   /* only used for the PTY driver */
18
19    /*
20     * Pointer to the tty data structures
21     */
22    struct tty_struct **ttys;
23    struct tty_port **ports;
24    struct ktermios **termios;
25    void *driver_state;
26
27    /*
28     * Driver methods
29     */
30
31    const struct tty_operations *ops;
32    struct list_head tty_drivers;
33 };

```

代码清单 12.18 tty\_operations 结构体

---

```

1 struct tty_operations {
2     struct tty_struct * (*lookup)(struct tty_driver *driver,
3             struct inode *inode, int idx);
4     int (*install)(struct tty_driver *driver, struct tty_struct *tty);
5     void (*remove)(struct tty_driver *driver, struct tty_struct *tty);
6     int (*open)(struct tty_struct * tty, struct file * filp);
7     void (*close)(struct tty_struct * tty, struct file * filp);
8     void (*shutdown)(struct tty_struct *tty);
9     void (*cleanup)(struct tty_struct *tty);
10    int (*write)(struct tty_struct * tty,
11                  const unsigned char *buf, int count);
12    int (*put_char)(struct tty_struct *tty, unsigned char ch);
13    void (*flush_chars)(struct tty_struct *tty);
14    int (*write_room)(struct tty_struct *tty);
15    int (*chars_in_buffer)(struct tty_struct *tty);
16    int (*ioctl)(struct tty_struct *tty,
17                  unsigned int cmd, unsigned long arg);
18    long (*compat_ioctl)(struct tty_struct *tty,
19                          unsigned int cmd, unsigned long arg);
20    void (*set_termios)(struct tty_struct *tty, struct ktermios * old);
21    void (*throttle)(struct tty_struct * tty);
22    void (*unthrottle)(struct tty_struct * tty);
23    void (*stop)(struct tty_struct *tty);
24    void (*start)(struct tty_struct *tty);
25    void (*hangup)(struct tty_struct *tty);
26    int (*break_ctl)(struct tty_struct *tty, int state);
27    void (*flush_buffer)(struct tty_struct *tty);
28    void (*set_ldisc)(struct tty_struct *tty);
29    void (*wait_until_sent)(struct tty_struct *tty, int timeout);
30    void (*send_xchar)(struct tty_struct *tty, char ch);
31    int (*tiocmget)(struct tty_struct *tty);
32    int (*tiocmset)(struct tty_struct *tty,
33                     unsigned int set, unsigned int clear);
34    int (*resize)(struct tty_struct *tty, struct winsize *ws);
35    int (*set_termiox)(struct tty_struct *tty, struct termiox *tnew);
36    int (*get_icount)(struct tty_struct *tty,
37                      struct serial_icounter_struct *icount);
38 #ifdef CONFIG_CONSOLE_POLL
39    int (*poll_init)(struct tty_driver *driver, int line, char *options);
40    int (*poll_get_char)(struct tty_driver *driver, int line);
41    void (*poll_put_char)(struct tty_driver *driver, int line, char ch);
42 #endif
43    const struct file_operations *proc_fops;
44 };

```

---

如图 12.10 所示, tty 设备发送数据的流程为: tty 核心从一个用户获取将要发送给一个 tty 设备的数据, tty 核心将数据传递给 tty 线路规程驱动, 接着数据被传递到 tty 驱动, tty 驱动将数据转换为可以发送给硬件的格式。接收数据的流程为: 从 tty 硬件接收到的数

据向上交给 tty 驱动，接着进入 tty 线路规程驱动，再进入 tty 核心，在这里它被一个用户获取。

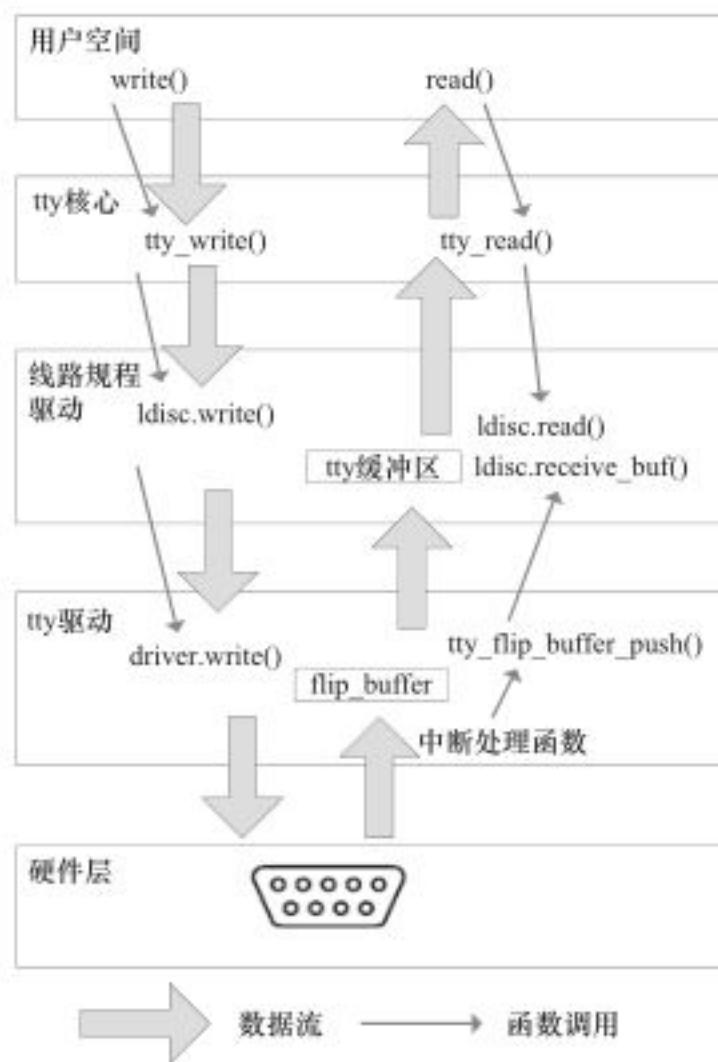


图 12.10 tty 设备发送、接收数据流的流程

代码清单 12.18 中第 10 行的 `tty_driver` 操作集 `tty_operations` 的成员函数 `write()` 函数接收 3 个参数：`tty_struct`、发送数据指针及要发送的字节数。该函数是被 `file_operations` 的 `write()` 成员函数间接触发调用的。从接收角度看，`tty` 驱动一般收到字符后会通过 `tty_flip_buffer_push()` 将接收缓冲区推到线路规程。

尽管一个特定的底层 UART 设备驱动完全可以遵循上述 `tty_driver` 的方法来设计，即定义 `tty_driver` 并实现 `tty_operations` 中的成员函数，但是鉴于串口之间的共性，Linux 考虑在文件 `drivers/tty/serial/serial_core.c` 中实现了 UART 设备的通用 `tty` 驱动层（我们可以称其为串口核心层）。这样，UART 驱动的主要任务就进一步演变成了实现 `serial-core.c` 中定义的一组 `uart_xxx` 接口而不是 `tty_xxx` 接口，如图 12.11 所示。因此，按照面向对象的思想，可以认为 `tty_driver` 是字符设备的泛化、`serial-core` 是 `tty_driver` 的泛化，而具体的串口驱动又是 `serial-core` 的泛化。

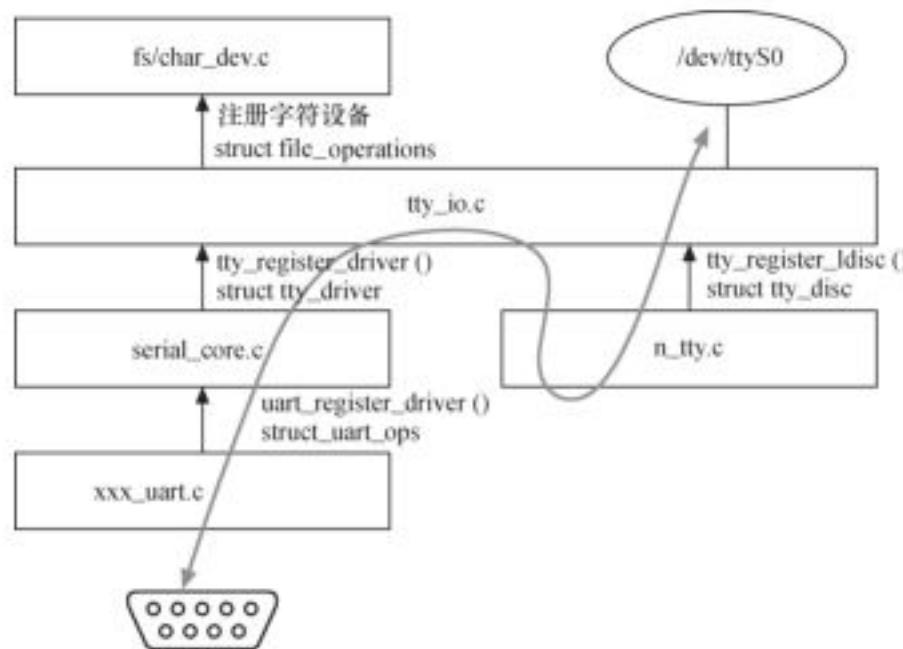


图 12.11 串口核心层

串口核心层又定义了新的 `uart_driver` 结构体和其操作集 `uart_ops`。一个底层的 UART 驱动需要创建并通过 `uart_register_driver()` 注册一个 `uart_driver` 而不是 `tty_driver`，代码清单 12.19 给出了 `uart_driver` 的定义。

代码清单 12.19 `uart_driver` 结构体

---

```

1 struct uart_driver {
2     struct module           *owner;
3     const char              *driver_name;
4     const char              *dev_name;
5     int                      major;
6     int                      minor;
7     int                      nr;
8     struct console          *cons;
9
10    /*
11     * these are private; the low level driver should not
12     * touch these; they should be initialised to NULL
13     */
14    struct uart_state        *state;
15    struct tty_driver         *tty_driver;
16 };
  
```

---

`uart_driver` 结构体在本质上是派生自 `uart_driver` 结构体，因此，它的第 15 行也包含了一个 `tty_driver` 结构体成员。`tty_operations` 在 UART 这个层面上也被进一步泛化为了 `uart_ops`，其定义如代码清单 12.20 所示。

代码清单 12.20 `uart_ops` 结构体

---

```

1 struct uart_ops {
2     unsigned int   (*tx_empty)(struct uart_port *);
  
```

---

```

3 void (*set_mctrl)(struct uart_port *, unsigned int mctrl);
4 unsigned int (*get_mctrl)(struct uart_port *);
5 void (*stop_tx)(struct uart_port *);
6 void (*start_tx)(struct uart_port *);
7 void (*throttle)(struct uart_port *);
8 void (*unthrottle)(struct uart_port *);
9 void (*send_xchar)(struct uart_port *, char ch);
10 void (*stop_rx)(struct uart_port *);
11 void (*enable_ms)(struct uart_port *);
12 void (*break_ctl)(struct uart_port *, int ctl);
13 int (*startup)(struct uart_port *);
14 void (*shutdown)(struct uart_port *);
15 void (*flush_buffer)(struct uart_port *);
16 void (*set_termios)(struct uart_port *, struct ktermios *new,
17                     struct ktermios *old);
18 void (*set_ldisc)(struct uart_port *, struct ktermios *);
19 void (*pm)(struct uart_port *, unsigned int state,
20            unsigned int oldstate);
21
22 const char *(*type)(struct uart_port *);
23
24 void (*release_port)(struct uart_port *);
25
26 int (*request_port)(struct uart_port *);
27 void (*config_port)(struct uart_port *, int);
28 int (*verify_port)(struct uart_port *, struct serial_struct *);
29 int (*ioctl)(struct uart_port *, unsigned int, unsigned long);
30 #ifdef CONFIG_CONSOLE_POLL
31 int (*poll_init)(struct uart_port *);
32 void (*poll_put_char)(struct uart_port *, unsigned char);
33 int (*poll_get_char)(struct uart_port *);
34#endif
35 );

```

---

由于 drivers/tty/serial/serial\_core.c 是一个 tty\_driver，因此在 serial\_core.c 中，存在一个 tty\_operations 的实例，这个实例的成员函数会进一步调用 struct uart\_ops 的成员函数，这样就把 file\_operations 里的成员函数、tty\_operations 的成员函数和 uart\_ops 的成员函数串起来了。

### 12.3.6 misc 设备驱动

由于 Linux 驱动倾向于分层设计，所以各个具体的设备都可以找到它归属的类型，从而套到它相应的架构里面去，并且只需要实现最底层的那一部分。但是，也有部分类似 globalmem、globalfifo 的字符设备，确实不知道它属于什么类型，我们一般推荐大家采用 miscdevice 框架结构。miscdevice 本质上也是字符设备，只是在 miscdevice 核心层的 misc\_init() 函数中，通过 register\_chrdev(MISC\_MAJOR, "misc", &misc\_fops) 注册了字符设备，而

具体 miscdevice 实例调用 misc\_register() 的时候又自动完成了 device\_create()、获取动态次设备号的动作。

miscdevice 的主设备号是固定的，MISC\_MAJOR 定义为 10，在 Linux 内核中，大概可以找到 200 多处使用 miscdevice 框架结构的驱动。

miscdevice 结构体的定义如代码清单 12.21 所示，在它的第 4 行，指向了一个 file\_operations 的结构体。miscdevice 结构体内 file\_operations 中的成员函数实际上是由 drivers/char/misc.c 中 misc 驱动核心层的 misc\_fops 成员函数间接调用的，比如 misc\_open() 就会间接调用底层注册的 miscdevice 的 fops->open。

代码清单 12.21 miscdevice 结构体

---

```

1 struct miscdevice {
2     int minor;
3     const char *name;
4     const struct file_operations *fops;
5     struct list_head list;
6     struct device *parent;
7     struct device *this_device;
8     const char *nodename;
9     umode_t mode;
10 };

```

---

如果上述代码第 2 行的 minor 为 MISC\_DYNAMIC\_MINOR，miscdevice 核心层会自动找一个空闲的次设备号，否则用 minor 指定的次设备号。第 3 行的 name 是设备的名称。

miscdevice 驱动的注册和注销分别用下面两个 API：

```

int misc_register(struct miscdevice * misc);
int misc_deregister(struct miscdevice *misc);

```

因此 miscdevice 驱动的一般结构形如：

```

static const struct file_operations xxx_fops = {
    .unlocked_ioctl = xxx_ioctl,
    .mmap           = xxx_mmap,
    ...
};

static struct miscdevice xxx_dev = {
    .minor   = MISC_DYNAMIC_MINOR,
    .name    = "xxx",
    .fops    = &xxx_fops
};

static int __init xxx_init(void)
{
    pr_info("ARC Hostlink driver mmap at 0x%p\n", __HOSTLINK__);
    return misc_register(&xxx_dev);
}

```

在调用 `misc_register(&xxx_dev)` 时，该函数内部会自动调用 `device_create()`，而 `device_create()` 会以 `xxx_dev` 作为 `drvdata` 参数。其次，在 `miscdevice` 核心层 `misc_open()` 函数的帮助下，在 `file_operations` 的成员函数中，`xxx_dev` 会自动成为 `file` 的 `private_data`（`misc_open` 会完成 `file->private_data` 的赋值操作）。

如果我们用面向对象的封装思想把一个设备的属性、自旋锁、互斥体、等待队列、`miscdevice` 等封装在一个结构体里面：

```
struct xxx_dev {
    unsigned int version;
    unsigned int size;
    spinlock_t lock;
    ...
    struct miscdevice miscdev;
};
```

在 `file_operations` 的成员函数中，就可以通过 `container_of()` 和 `file->private_data` 反推出 `xxx_dev` 的实例。

```
static long xxx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    struct xxx_dev *xxx = container_of(file->private_data,
                                         struct xxx_dev, miscdev);
    ...
}
```

下面我们把 `globalfifo` 驱动改造成基于 `platform_driver` 且采用 `miscdevice` 框架的结构体。首先这个新的驱动变成了要通过 `platform_driver` 的 `probe()` 函数来初始化，其次不再直接采用 `register_chrdev()`、`cdev_add()` 之类的原始 API，而采用 `miscdevice` 的注册方法。代码清单 12.22 列出了新的 `globalfifo` 驱动相对于第 9 章 `globalfifo` 驱动变化的部分。

代码清单 12.22 新的 `globalfifo` 驱动相对于第 9 章 `globalfifo` 驱动变化的部分

---

```
1 struct globalfifo_dev {
2     ...
3     struct miscdevice miscdev;
4 };
5
6 static int globalfifo_fasync(int fd, struct file *filp, int mode)
7 {
8     struct globalfifo_dev *dev = container_of(filp->private_data,
9         struct globalfifo_dev, miscdev);
10    ...
11 }
12
13 static long globalfifo_ioctl(struct file *filp, unsigned int cmd,
14                             unsigned long arg)
15 {
```

```
16     struct globalfifo_dev *dev = container_of(filp->private_data,
17         struct globalfifo_dev, miscdev);
18     ...
19 }
20
21 static unsigned int globalfifo_poll(struct file *filp, poll_table * wait)
22 {
23     struct globalfifo_dev *dev = container_of(filp->private_data,
24         struct globalfifo_dev, miscdev);
25     ...
26 }
27
28 static ssize_t globalfifo_read(struct file *filp, char __user *buf,
29             size_t count, loff_t *ppos)
30 {
31     struct globalfifo_dev *dev = container_of(filp->private_data,
32         struct globalfifo_dev, miscdev);
33     ...
34 }
35
36 static ssize_t globalfifo_write(struct file *filp, const char __user *buf,
37             size_t count, loff_t *ppos)
38 {
39     struct globalfifo_dev *dev = container_of(filp->private_data,
40         struct globalfifo_dev, miscdev);
41     ...
42 }
43
44 static int globalfifo_probe(struct platform_device *pdev)
45 {
46     struct globalfifo_dev *gl;
47     int ret;
48
49     gl = devm_kzalloc(&pdev->dev, sizeof(*gl), GFP_KERNEL);
50     if (!gl)
51         return -ENOMEM;
52     gl->miscdev.minor = MISC_DYNAMIC_MINOR;
53     gl->miscdev.name = "globalfifo";
54     gl->miscdev.fops = &globalfifo_fops;
55
56     mutex_init(&gl->mutex);
57     init_waitqueue_head(&gl->r_wait);
58     init_waitqueue_head(&gl->w_wait);
59     platform_set_drvdata(pdev, gl);
60
61     ret = misc_register(&gl->miscdev);
62     if (ret < 0)
63         goto err;
64     ...
65     return 0;
66 err:
```

```

67     return ret;
68 }
69
70 static int globalfifo_remove(struct platform_device *pdev)
71 {
72     struct globalfifo_dev *gl = platform_get_drvdata(pdev);
73     misc_deregister(&gl->miscdev);
74     return 0;
75 }
76
77 static struct platform_driver globalfifo_driver = {
78     .driver = {
79         .name = "globalfifo",
80         .owner = THIS_MODULE,
81     },
82     .probe = globalfifo_probe,
83     .remove = globalfifo_remove,
84 };
85 module_platform_driver(globalfifo_driver);

```

---

在上述代码中，file\_operations 的各个成员函数都使用 container\_of() 反向求出 private\_data，第 61 行在 platform\_driver 的 probe() 函数中完成了 miscdev 的注册，而在 remove() 函数中使用 misc\_deregister() 完成了 miscdev 的注销。

上述代码也改用了 platform\_device 和 platform\_driver 的体系结构。我们增加了一个模块来完成 platform\_device 的注册，在模块初始化的时候通过 platform\_device\_alloc() 和 platform\_device\_add() 分配并添加 platform\_device，而在模块卸载的时候则通过 platform\_device\_unregister() 注销 platform\_device，如代码清单 12.23 所示。

代码清单 12.23 与 globalfifo 对应的 platform\_device 的注册和注销

```

1 static struct platform_device *globalfifo_pdev;
2
3 static int __init globalfifodev_init(void)
4 {
5     int ret;
6
7     globalfifo_pdev = platform_device_alloc("globalfifo", -1);
8     if (!globalfifo_pdev)
9         return -ENOMEM;
10
11    ret = platform_device_add(globalfifo_pdev);
12    if (ret) {
13        platform_device_put(globalfifo_pdev);
14        return ret;
15    }
16
17    return 0;
18

```

```

19 }
20 module_init(globalfifo_dev_init);
21
22 static void __exit globalfifo_dev_exit(void)
23 {
24     platform_device_unregister(globalfifo_pdev);
25 }
26 module_exit(globalfifo_dev_exit);

```

本书配套代码 /home/baohua/develop/training/kernel/drivers/globalfifo/ch12 中包含了 globalfifo driver 和 device 端的两个模块。在该目录运行 make，会生成两个模块：globalfifo.ko 和 globalfifo-dev.ko，把 globalfifo.ko 和 globalfifo-dev.ko 先后 insmod，会导致 platform\_driver 和 platform\_device 的匹配，globalfifo\_probe() 会执行，/dev/globalfifo 节点会自动生成，默认情况下需要 root 权限来访问 /dev/globalfifo。

如果此后我们 rmmod globalfifo-dev.ko，则会导致 platform\_driver 的 remove() 成员函数，即 globalfifo\_remove() 函数被执行，/dev/globalfifo 节点会自动消失。

### 12.3.7 驱动核心层

分析了上述多个实例，我们可以归纳出核心层肩负的 3 大职责：

- 1) 对上提供接口。file\_operations 的读、写、ioctl 都被中间层搞定，各种 I/O 模型也被处理掉了。

- 2) 中间层实现通用逻辑。可以被底层各种实例共享的代码都被中间层搞定，避免底层重复实现。

- 3) 对下定义框架。底层的驱动不再需要关心 Linux 内核 VFS 的接口和各种可能的 I/O 模型，而只需处理与具体硬件相关的访问。

这种分层有时候还不是两层，可以有更多层，在软件上呈现为面向对象里类继承和多态的状态。上一节介绍的终端设备驱动，在软件层次上类似图 12.12 的效果。

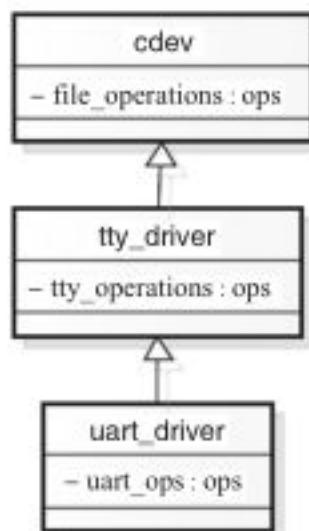


图 12.12 tty 驱动各层泛化

## 12.4 主机驱动与外设驱动分离的设计思想

### 12.4.1 主机驱动与外设驱动分离

Linux 中的 SPI、I<sup>2</sup>C、USB 等子系统都利用了典型的把主机驱动和外设驱动分离的想法，让主机端只负责产生总线上的传输波形，而外设端只是通过标准的 API 来让主机端以适当的波形访问自身。因此这里面就涉及了 4 个软件模块：

1) 主机端的驱动。根据具体的I<sup>2</sup>C、SPI、USB等控制器的硬件手册，操作具体的I<sup>2</sup>C、SPI、USB等控制器，产生总线的各种波形。

2) 连接主机和外设的纽带。外设不直接调用主机端的驱动来产生波形，而是调一个标准的API。由这个标准的API把这个波形的传输请求间接“转发”给了具体的主机端驱动。当然，在这里，最好把关于波形的描述也以某种数据结构标准化。

3) 外设端的驱动。外设接在I<sup>2</sup>C、SPI、USB这样的总线上，但是它们本身可以是触摸屏、网卡、声卡或者任意一种类型的设备。我们在相关的i2c\_driver、spi\_driver、usb\_driver这种xxx\_driver的probe()函数中去注册它具体的类型。当这些外设要求I<sup>2</sup>C、SPI、USB等去访问它的时候，它调用“连接主机和外设的纽带”模块的标准API。

4) 板级逻辑。板级逻辑用来描述主机和外设是如何互联的，它相当于一个“路由表”。假设板子上有多个SPI控制器和多个SPI外设，那究竟谁接在谁上面？管理互联关系，既不是主机端的责任，也不是外设端的责任，这属于板级逻辑的责任。这部分通常出现在arch/arm/mach-xxx下面或者arch/arm/boot/dts下面。

什么叫良好的软件设计？一言以蔽之，让正确的代码出现在正确的位置。不要在错误的时间、错误的地点，编写一段错误的代码。在LKML中，关于代码出现在错误的位置，常见的台词是代码“out of place”。

Linux通过上述的设计方法，把一堆杂乱不友好的代码变成了4个轻量级的小模块，每个模块都各得其所。每个模块都觉得很“爽”，站在主机端想一想，它其实也是很“爽”的，因为它的职责本来就是产生波形，而现在我们就让它只产生波形不干别的；站在外设端想一想，它也变得一身轻松，因为它根本就不需要知道自己接在主机的哪个控制器上，根本不关心对方是张三、李四、王五还是六麻子；站在板级逻辑的角度上，你做了一个板子，自己自然要知道谁接在谁上面了。

下面以SPI子系统为例来展开说明，后续章节的I<sup>2</sup>C、USB等是类似的。

### 12.4.2 Linux SPI主机和设备驱动

在Linux中，用代码清单12.24的spi\_master结构体来描述一个SPI主机控制器驱动，其主要成员是主机控制器的序号（系统中可能存在多个SPI主机控制器）、片选数量、SPI模式、时钟设置用到的和数据传输用到的函数等。

代码清单12.24 spi\_master结构体

---

```

1 struct spi_master {
2     struct devicedev;
3
4     s16          bus_num;
5
6     /* chipselects will be integral to many controllers; some others
7      * might use board-specific GPIOs.
8      */

```

```
9    u16          num_chipselect;
10
11 ...
12
13 /* limits on transfer speed */
14 u32          min_speed_hz;
15 u32          max_speed_hz;
16
17 ...
18
19 /* Setup mode and clock, etc (spi driver may call many times).
20 *
21 * IMPORTANT: this may be called when transfers to another
22 * device are active. DO NOT UPDATE SHARED REGISTERS in ways
23 * which could break those transfers.
24 */
25 int          (*setup)(struct spi_device *spi);
26
27 /* bidirectional bulk transfers
28 *
29 * + The transfer() method may not sleep; its main role is
30 *   just to add the message to the queue.
31 * + For now there's no remove-from-queue operation, or
32 *   any other request management
33 * + To a given spi_device, message queueing is pure fifo
34 *
35 * + The master's main job is to process its message queue,
36 *   selecting a chip then transferring data.
37 * + If there are multiple spi_device children, the i/o queue
38 *   arbitration algorithm is unspecified (round robin, fifo,
39 *   priority, reservations, preemption, etc)
40 *
41 * + Chipselect stays active during the entire message
42 *   (unless modified by spi_transfer.cs_change != 0).
43 * + The message transfers use clock and SPI mode parameters
44 *   previously established by setup() for this device
45 */
46 int          (*transfer)(struct spi_device *spi,
47                      struct spi_message *mesg);
48
49 /* called on release() to free memory provided by spi_master */
50 void         (*cleanup)(struct spi_device *spi);
51
52 ...
53 /*
54 * These hooks are for drivers that use a generic implementation
55 * of transfer_one_message() provided by the core.
56 */
57
58 void (*set_cs)(struct spi_device *spi, bool enable);
59 int (*transfer_one)(struct spi_master *master, struct spi_device *spi,
```

---

```

60             struct spi_transfer *transfer);
61
62     /* gpio chip select */
63     int          *cs_gpios;
64
65     ...
66 };

```

---

分配、注册和注销 SPI 主机的 API 由 SPI 核心提供：

```

struct spi_master * spi_alloc_master(struct device *host, unsigned size);
int spi_register_master(struct spi_master *master);
void spi_unregister_master(struct spi_master *master);

```

在 Linux 中，用代码清单 12.25 的 `spi_driver` 结构体来描述一个 SPI 外设驱动，这个外设驱动可以认为是 `spi_master` 的客户端驱动。

代码清单 12.25 `spi_driver` 结构体

---

```

1 struct spi_driver {
2     const struct spi_device_id *id_table;
3     int          (*probe)(struct spi_device *spi);
4     int          (*remove)(struct spi_device *spi);
5     void         (*shutdown)(struct spi_device *spi);
6     int          (*suspend)(struct spi_device *spi, pm_message_t mesg);
7     int          (*resume)(struct spi_device *spi);
8     struct device_driver    driver;
9 };

```

---

可以看出，`spi_driver` 结构体和 `platform_driver` 结构体有极大的相似性，都有 `probe()`、`remove()`、`suspend()`、`resume()` 这样的接口和 `device_driver` 的实例。是的，这几乎是一切客户端驱动的常用模板。

在 SPI 外设驱动中，当通过 SPI 总线进行数据传输的时候，使用了一套与 CPU 无关的统一的接口。这套接口的第一个关键数据结构就是 `spi_transfer`，它用于描述 SPI 传输，如代码清单 12.26 所示。

代码清单 12.26 `spi_transfer` 结构体

---

```

1 struct spi_transfer {
2     /* it's ok if tx_buf == rx_buf (right?)
3      * for MicroWire, one buffer must be null
4      * buffers must work with dma_*map_single() calls, unless
5      *   spi_message.is_dma_mapped reports a pre-existing mapping
6      */
7     const void    *tx_buf;
8     void          *rx_buf;
9     unsigned      len;
10
11    dma_addr_t   tx_dma;

```

---

```

12  dma_addr_t      rx_dma;
13  struct sg_table tx_sg;
14  struct sg_table rx_sg;
15
16  unsigned        cs_change:1;
17  unsigned        tx_nb_bits:3;
18  unsigned        rx_nb_bits:3;
19 #define      SPI_NBITS_SINGLE          0x01 /* 1bit transfer */
20 #define      SPI_NBITS_DUA            0x02 /* 2bits transfer */
21 #define      SPI_NBITS_QUAD           0x04 /* 4bits transfer */
22  u8              bits_per_word;
23  u16             delay_usecs;
24  u32             speed_hz;
25
26  struct list_head transfer_list;
27 };
```

而一次完整的 SPI 传输流程可能不是只包含一次 spi\_transfer，它可能包含一个或多个 spi\_transfer，这些 spi\_transfer 最终通过 spi\_message 组织在一起，其定义如代码清单 12.27 所示。

代码清单 12.27 spi\_message 结构体

```

1  struct spi_message {
2  struct list_head    transfers;
3
4  struct spi_device   *spi;
5
6  unsigned           is_dma_mapped:1;
7
8  /* REVISIT: we might want a flag affecting the behavior of the
9   * last transfer ... allowing things like "read 16 bit length L"
10  * immediately followed by "read L bytes". Basically imposing
11  * a specific message scheduling algorithm.
12  *
13  * Some controller drivers (message-at-a-time queue processing)
14  * could provide that as their default scheduling algorithm. But
15  * others (with multi-message pipelines) could need a flag to
16  * tell them about such special cases.
17  */
18
19  /* completion is reported through a callback */
20  void              (*complete)(void *context);
21  void              *context;
22  unsigned          frame_length;
23  unsigned          actual_length;
24  int               status;
25
26  /* for optional use by whatever driver currently owns the
27  * spi_message ... between calls to spi_async and then later
```

---

```

28     * complete(), that's the spi_master controller driver.
29     */
30     struct list_head    queue;
31     void                *state;
32 };

```

---

通过 spi\_message\_init() 可以初始化 spi\_message，而将 spi\_transfer 添加到 spi\_message 队列的方法则是：

```
void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m);
```

发起一次 spi\_message 的传输有同步和异步两种方式，使用同步 API 时，会阻塞等待这个消息被处理完。同步操作时使用的 API 是：

```
int spi_sync(struct spi_device *spi, struct spi_message *message);
```

使用异步 API 时，不会阻塞等待这个消息被处理完，但是可以在 spi\_message 的 complete 字段挂接一个回调函数，当消息被处理完成后，该函数会被调用。在异步操作时使用的 API 是：

```
int spi_async(struct spi_device *spi, struct spi_message *message);
```

代码清单 12.28 是非常典型的初始化 spi\_transfer、spi\_message 并进行 SPI 数据传输的例子，同时 spi\_write()、spi\_read() 也是 SPI 核心层的两个通用快捷 API，在 SPI 外设驱动中可以直接调用它们进行简单的纯写和纯读操作。

**代码清单 12.28 SPI 传输实例 spi\_write()、spi\_read() API**

---

```

1 static inline int
2 spi_write(struct spi_device *spi, const u8 *buf, size_t len)
3 {
4     struct spi_transfer      t = {
5         .tx_buf            = buf,
6         .len               = len,
7     };
8     struct spi_message      m;
9
10    spi_message_init(&m);
11    spi_message_add_tail(&t, &m);
12    return spi_sync(spi, &m);
13 }
14
15 static inline int
16 spi_read(struct spi_device *spi, u8 *buf, size_t len)
17 {
18     struct spi_transfer      t = {
19         .rx_buf            = buf,
20         .len               = len,
21     };

```

---

```

22     struct spi_message      m;
23
24     spi_message_init(&m);
25     spi_message_add_tail(&t, &m);
26     return spi_sync(spi, &m);
27 }

```

SPI 主机控制器驱动位于 drivers/spi/，这些驱动的主体是实现了 spi\_master 的 transfer()、transfer\_one()、setup() 这样的成员函数，当然，也可能是实现 spi\_bitbang 的 txrx\_bufs()、setup\_transfer()、chipselect() 这样的成员函数。代码清单 12.29 摘取了 drivers/spi/spi-pl022.c 的部分代码。

代码清单 12.29 SPI 主机端驱动完成的波形传输

```

1 static int pl022_transfer_one_message(struct spi_master *master,
2                                     struct spi_message *msg)
3 {
4     struct pl022 *pl022 = spi_master_get_devdata(master);
5
6     /* Initial message state */
7     pl022->cur_msg = msg;
8     msg->state = STATE_START;
9
10    pl022->cur_transfer = list_entry(msg->transfers.next,
11                                       struct spi_transfer, transfer_list);
12
13    /* Setup the SPI using the per chip configuration */
14    pl022->cur_chip = spi_get_ctldata(msg->spi);
15    pl022->cur_cs = pl022->chipselects[msg->spi->chip_select];
16
17    restore_state(pl022);
18    flush(pl022);
19
20    if (pl022->cur_chip->xfer_type == POLLING_TRANSFER)
21        do_polling_transfer(pl022);
22    else
23        do_interrupt_dma_transfer(pl022);
24
25    return 0;
26 }
27
28 static int pl022_setup(struct spi_device *spi)
29 {
30     ...
31     /* Stuff that is common for all versions */
32     if (spi->mode & SPI_CPOL)
33         tmp = SSP_CLK_POL_IDLE_HIGH;
34     else
35         tmp = SSP_CLK_POL_IDLE_LOW;

```

```

36     SSP_WRITE_BITS(chip->cr0, tmp, SSP_CRO_MASK_SPO, 6);
37
38     if (spi->mode & SPI_CPHA)
39         tmp = SSP_CLK_SECOND_EDGE;
40     else
41         tmp = SSP_CLK_FIRST_EDGE;
42     SSP_WRITE_BITS(chip->cr0, tmp, SSP_CRO_MASK_SPH, 7);
43
44     ...
45 }
46
47 static int p1022_probe(struct amba_device *adev, const struct amba_id *id)
48 {
49     ...
50
51     /*
52      * Bus Number Which has been Assigned to this SSP controller
53      * on this board
54      */
55     master->bus_num = platform_info->bus_id;
56     master->num_chipselect = num_cs;
57     master->cleanup = p1022_cleanup;
58     master->setup = p1022_setup;
59     master->auto_runtime_pm = true;
60     master->transfer_one_message = p1022_transfer_one_message;
61     master->unprepare_transfer_hardware = p1022_unprepare_transfer_hardware;
62     master->rt = platform_info->rt;
63     master->dev.of_node = dev->of_node;
64
65     ...
66 }

```

SPI 外设驱动遍布于内核的 drivers、sound 的各个子目录之下，SPI 只是一种总线， spi\_driver 的作用只是将 SPI 外设挂接在该总线上，因此在 spi\_driver 的 probe() 成员函数中，将注册 SPI 外设本身所属设备驱动的类型。

和 platform\_driver 对应着一个 platform\_device 一样， spi\_driver 也对应着一个 spi\_device； platform\_device 需要在 BSP 的板文件中添加板信息数据，而 spi\_device 也同样需要。 spi\_device 的板信息用 spi\_board\_info 结构体描述，该结构体记录着 SPI 外设使用的主机控制器序号、片选序号、数据比特率、SPI 传输模式（即 CPOL、CPHA）等。诺基亚 770 上的两个 SPI 设备的板信息数据如代码清单 12.30 所示，位于板文件 arch/arm/mach-omap1/board-nokia770.c 中。

代码清单 12.30 诺基亚 770 板文件中的 spi\_board\_info

---

```

1 static struct spi_board_info nokia770_spi_board_info[] __initdata = {
2     [0] = {
3         .modalias           = "lcd_mipid",

```

```

4     .bus_num          = 2,           /* 用到的 SPI 主机控制器序号 */
5     .chip_select      = 3,           /* 使用哪个片选 */
6     .max_speed_hz    = 12000000,   /* SPI 数据传输比特率 */
7     .platform_data    = &nokia770_mipid_platform_data,
8   },
9   [1] = {
10    .modalias         = "ads7846",
11    .bus_num          = 2,
12    .chip_select      = 0,
13    .max_speed_hz    = 2500000,
14    .irq              = OMAP_GPIO_IRQ(15),
15    .platform_data    = &nokia770_ads7846_platform_data,
16  },
17 };

```

在 Linux 启动过程中，在机器的 init\_machine() 函数中，会通过如下语句注册这些 spi\_board\_info：

```
spi_register_board_info(nokia770_spi_board_info,
                        ARRAY_SIZE(nokia770_spi_board_info));
```

这一点和启动时通过 platform\_add\_devices() 添加 platform\_device 非常相似。

ARM Linux 3.x 之后的内核在改为设备树后，不再需要在 arch/arm/mach-xxx 中编码 SPI 的板级信息了，而倾向于在 SPI 控制器节点下填写子节点，如代码清单 12.31 给出了 arch/arm/boot/dts/omap3-overo-common-lcd43.dtsi 中包含的 ads7846 节点。

代码清单 12.31 通过设备树添加 SPI 外设

```

1 &mcspl {
2   pinctrl-names = "default";
3   pinctrl-0 = <&mcspl_pins>;
4
5   /* touch controller */
6   ads7846@0 {
7     pinctrl-names = "default";
8     pinctrl-0 = <&ads7846_pins>;
9
10    compatible = "ti,ads7846";
11    vcc-supply = <&ads7846reg>;
12
13    reg = <0>;                                /* CS0 */
14    spi-max-frequency = <1500000>;
15
16    interrupt-parent = <&gpio4>;
17    interrupts = <18 0>;                      /* gpio_114 */
18    pendown-gpio = <&gpio4 18 0>;
19
20    ti,x-min = /bits/ 16 <0x0>;
21    ti,x-max = /bits/ 16 <0xffff>;

```

```
22          ti,y-min = /bits/ 16 <0x0>;
23          ti,y-max = /bits/ 16 <0xffff>;
24          ti,x-plate-ohms = /bits/ 16 <180>;
25          ti,pressure-max = /bits/ 16 <255>;
26
27          linux,wakeup;
28      };
29  };
```

---

## 12.5 总结

现实生活中的驱动并不像第6~11章里那样的驱动，它往往包含了platform、分层、分离等诸多概念，因此，学习和领悟第12章的内容是我们将驱动的理论用于工程开发的必要环节。Linux内核目前有百多个驱动子系统，一个个去学肯定是不现实的，在方法上也是错误的。我们要掌握其规律，以不变应万变，以无招胜有招。