第十五章 Android 内核驱动——杂项

15.1 日志系统

基本原理

Android 的 logger 是一种轻量级的日志系统。在内核中实现为一种 misc 设备驱动，它与用户态的 logcat 工具配合实现了方便的调试工具，开发应用程序的时候可以利用 logger 查看日志，进行跟踪调试。

logger 的实现

logger 的源代码在 drivers/staging/android/logger.c 中，它用三个结构体 logger_log，logger_reader 和 logger_entry 来维护 logger 设备的信息。其中 logger_log 代表一个 log 设备，logger_reader 代表一个读日志的 reader，logger_entry 代表 writer 写入的一条日志。

logger 在模块初始化时注册三个 misc 设备：log_main，log_events 和 log_radio。其中 log_main 记录主要的日志信息，log_events 记录与事件有关的信息，log_radio 记录与通信有关的信息，实现了以下的 file operation：

- logger_open

标准的 open 接口，如果以读模式打开，则分配一个 logger_reader，初始化其成员变量，并把这个 logger_reader 保存在 file->private_data 中。如果是 write 模式打开，则直接把对应 logger 设备的 logger_log 保存在 file->private_data。此后在读或者写的时候就可以通过 file->private_data 找到对应的 logger_reader 或 logger_log。

- logger_read

首先当前进程（读 log 的进程）加到 logger_log->wq 等待队列上，判断当前日志 buffer 是否为空，如果空则调度别的进程运行，自己挂起（如果指定了非阻塞模式，则直接返回-EAGAIN），重复上述过程直 buffer 中有日志可读，此时，读出一条日志，拷贝到用户空间，返回。

- logger_aio_write

写操作支持同步、异步以及 scatter 方式的写操作。写操作几乎总是成功的，当 buffer 满的时候，新写入的日志会覆盖最初的日志。总之，buffer 是环形的，如果没有及时被读出，数据会丢失。

- logger_ioctl

支持以下命令：

  - LOGGER_GET_LOG_BUF_SIZE：得到 logger device 环形缓冲区的大小

- LOGGER_GET_LOG_LEN：得到当前日志 buffer 中未被读出的日志长度
  LOGGER_GET_NEXT_ENTRY_LEN：得到下一条日志长度(即紧接着上次读出的日志后面一条)
- LOGGER_FLUSH_LOG：清空日志

- logger_poll

查询当前进程是否可以对 logger device 操作。POLLOUT 总是成立的，即进程总是可以写入日志。但只有以 FMODE_READ 模式打开 logger 设备的进程，并且当前日志非空，才可以读到日志。

用户接口

logger 日志系统是标准的 misc 设备，提供标准的 file operation，应用程序可以通过标准的 C 库文件函数操作日志系统。

在 Android 应用开发中，比较有用的是 logcat 命令，通过该命令可以查看系统的日志输出，在调试的时候，应用程序插入日志，在 logcat 中就可以看到，这样就实现了方便的插桩跟踪调试。

logcat 使用方法如下：

```
logcat [options] [filterspecs]
```

logcat 的选项包括：

-s	设置过滤器，例如指定 '*：s'
-f <filename>	输出到文件，默认情况是标准输出。
-r [<kbytes>]	Rotate log every kbytes. (16 if unspecified). Requires –f
-n <count>	Sets max number of rotated logs to <count>， default 4
-v <format>	设置 log 的打印格式， <format> 是下面的一种： brief process tag thread raw time threadtime long
-c	清除所有 log 并退出
-d	得到所有 log 并退出 (不阻塞)
-g	得到环形缓冲区的大小并退出
-b <buffer>	请求不同的环形缓冲区 ('main'(默认)， 'radio'， 'events')
-B	输出 log 到二进制中

过滤器的格式是一个这样的串：

```
 <tag>[: priority]
```

其中<tag>表示 log 的 component，tag（或者使用 * 表示所有），可以应用中定义，这样在用 logcat 查看日志的时候就可以指定这个 tag，只查看这个 tag 的应用产生的日志。

priority 如下所示：

V Verbose

D Debug

I Info
W Warn
E Error
F Fatal
S Silent

事实上 logcat 的功能是由 Android 的类 android.util.Log 决定的，在程序中 log 的使用方法如下所示：

```
Log.v() //-------------------- VERBOSE
Log.d() //-------------------- DEBUG
Log.i() //-------------------- INFO
Log.w() //-------------------- WARN
Log.e() //-------------------- ERROR
```

以上 log 的级别依次升高，DEBUG 信息应当只存在于开发中，INFO， WARN， ERROR 这三种 log 将出现在发布版本中。

15.2 Switch

基本原理

Switch 是 Android 引进的新的驱动，目的是用于检测一些开关量，比如检测耳机插入、检测 USB 设备插入等。Switch 在 sysfs 文件系统中创建相应 entry，用户可以通过 sysfs 与之交互；此外还可以通过 uevent 机制与之交互， 从而检测 switch 状态。

Switch 的实现

Switch class 在 Android 中实现为一个 module，可动态加载；而具体的 switch gpio 则是基于 platform device 框架。代码在 drivers\switch\switch_class.c 和 drivers\switch\switch_gpio.c 中。其中 switch_class.c 实现了一个 switch class，而 switch_gpio.c 则是这个 class 中的一个 device，即针对 gpio 的一个 switch 设备。

switch_class.c 文件创建了一个 switch_class，实现了内核的 switch 机制，提供支持函数供其他 switch device 驱动调用。

```
static int __init switch_class_init(void){
     return create_switch_class();
}

static void __exit switch_class_exit(void){
     class_destroy(switch_class);
}

module_init(switch_class_init);
module_exit(switch_class_exit);
```

init 函数调用 create_switch_class->class_create 创建 switch_class 设备类。相对应 exit 则是销毁这个设备类。

该文件导出两个函数供其他 switch 设备驱动调用，分别是注册 switch 设备 switch_dev_register 和注销 switch_dev_unregister。

```
int switch_dev_register(struct switch_dev *sdev)
{
      int ret;
      //如果 switch_class 还未创建,则创建它
      if (!switch_class) {
           ret = create_switch_class();
           if (ret < 0)
                return ret;
      }
      // 创建这个 switch device 的设备对象,保存在其 struct switch_dev->dev 中
      sdev->index = atomic_inc_return(&device_count);
      sdev->dev = device_create(switch_class, NULL,
           MKDEV(0, sdev->index), NULL, sdev->name);
      if (IS_ERR(sdev->dev))
           return PTR_ERR(sdev->dev);
      //在 sysfs 中分别创建两个 entry,一个用于输出设备 state,一个是输出设备名称
      ret = device_create_file(sdev->dev, &dev_attr_state);
      if (ret < 0)
           goto err_create_file_1;
      ret = device_create_file(sdev->dev, &dev_attr_name);
      if (ret < 0)
           goto err_create_file_2;
      dev_set_drvdata(sdev->dev, sdev);
      sdev->state = 0;
      return 0;
err_create_file_2:
      device_remove_file(sdev->dev, &dev_attr_state);
err_create_file_1:
      device_destroy(switch_class, MKDEV(0, sdev->index));
      printk(KERN_ERR "switch:  Failed to register driver %s\n", sdev->name);
      return ret;
}
EXPORT_SYMBOL_GPL(switch_dev_register);
```

```
void switch_dev_unregister(struct switch_dev *sdev)
{
    device_remove_file(sdev->dev, &dev_attr_name);
    device_remove_file(sdev->dev, &dev_attr_state);
    device_destroy(switch_class, MKDEV(0, sdev->index));
    dev_set_drvdata(sdev->dev, NULL);
}
EXPORT_SYMBOL_GPL(switch_dev_unregister);
```

然后是两个 sysfs 操作函数(state_show 和 name_show)，分别用于输出 switch device 的 name 和 state。当用户读取 sysfs 中对应的 switch entry（/sys/class/switch/<dev_name>/name 和 /sys/class/switch/<dev_name>/state）时候，系统会自动调用这两个函数向用户返回 switch 设备的名称和状态。

```
static ssize_t state_show(struct device *dev, struct device_attribute *attr,char
*buf)
{
     struct switch_dev *sdev = (struct switch_dev *)
          dev_get_drvdata(dev);
```

```
    if (sdev->print_state) {
          int ret = sdev->print_state(sdev, buf);
          if (ret >= 0)
               return ret;
     }
     return sprintf(buf, "%d\n", sdev->state);
}
```

```
static ssize_t name_show(struct device *dev, struct device_attribute *attr,char
*buf)
{
     struct switch_dev *sdev = (struct switch_dev *)
          dev_get_drvdata(dev);
     if (sdev->print_name) {
          int ret = sdev->print_name(sdev, buf);
          if (ret >= 0)
               return ret;
     }
     return sprintf(buf, "%s\n", sdev->name);
}
```

可见，这两个函数就是直接调用对应的 switch_dev 中的 print_state 和 print_name 函数；如果没有定义这两个函数，则调用 sprintf 把信息打印到 buf 缓冲区里。

最后是 switch_set_state 函数，该函数是内核内部使用，并不为用户调用，它完成的功能主要是两件事：

- 调用 name_show 和 state_show 输出 switch 设备名称和状态至 sysfs 文件系统
- 发送 uevent 通知用户 switch device 的信息(名称和状态)

switch_gpio.c 文件基于 switch class 实现了一个 gpio 的 switch 设备驱动，其实现的原理如下：

基于 platform device/driver 框架，在 probe 函数中完成初始化，包括获取 gpio 的使用权限，设置 gpio 方向为输入，注册 switch_dev 设备，为 gpio 分配中断，指定中断服务程序，初始化一个 gpio_switch_work 工作，最后读取 gpio 初始状态。

当 GPIO 引脚状态发生变化时，则会触发中断，在中断服务程序中调用 schedule_work，这个被 schedule 的 work 即前面初始化的 gpio_switch_work，最后这个 work 被执行，在 gpio_switch_work 函数中读取当前 gpio 电平，调用 switch_set_state 更新 sysfs 并通过 uevent 通知上层应用。

这个设备驱动只实现了 print_state 函数：switch_gpio_print_state，没有实现 print_name 函数。当 gpio_switch_work 执行的时候，里面调用 switch_set_state->switch_gpio_print_state 输出 GPIO 状态到 sysfs。

用户接口

sysfs 文件系统和 uevent 机制。sysfs 文件为 sys/class/switch/<dev_name>/name，sys/class/switch/<dev_name>/state，uevent 环境变量为 SWITCH_NAME=<name>，SWITCH_STATE=<state>。

15.3 Timed GPIO

基本原理

Timed GPIO 基于 platform driver 实现了一个增强的 GPIO 驱动。与普通 GPIO 驱动不同的地方就是 Timed GPIO 将普通 GPIO 与内核定时器绑定在一起，实现了一种时钟控制的 GPIO；当定时器过期后，GPIO 的状态会设置为指定的状态。

Android 的 timed GPIO 实际上实现的功能是：通过 sysfs 操作 GPIO，比如可以让 GPIO 输出高/低电平；但同时可以指定一个定时器过期时间。到达过期时间后，执行一个 callback 函数，可以重新设置 GPIO 的输出电平。

Timed GPIO 的实现

Timed GPIO 代码在 drivers/staging/android 下，对应的文件为：timed_gpio.c， timed_gpio.h，timed_output.c，timed_output.h。Timed GPIO 驱动实现为 platform driver，在其 probe 函数中， sysfs 创建了相应的设备文件和 class 文件，应用程序可以通过设备文件实现与内核驱动的交互。

timed_output.c 实现了一个 timed output class driver，该文件主要是创建了一个名为 timed_output 的设备 class，实现 sysfs 相关功能，提供 show，store 两个函数以供应用层调用。Timed_output.c 还提供了用于注册和注销 timed GPIO 设备到 linux 设备框架的函数，具体说就是创建/删除对应的设备文件，创建/删除用于表示设备的 struct device 对象。所有的 timed GPIO 设备都是属于这个 timed_output class 的。

Timed_gpio.c 文件则实现了一个 timed GPIO driver，它基于 timed_output.c 提供的功能，实现了一个基于 platform driver 架构的驱动，提供的也是标准的接口。其 init 函数和 exit 函数分别调用 platform_driver_register 和 platform_driver_unregister 注册/注销 timed_gpio_driver。

```
static struct platform_driver timed_gpio_driver = {
     .probe         = timed_gpio_probe,
     .remove        = timed_gpio_remove,
     .driver        ={
         .name          = TIMED_GPIO_NAME,
     .owner = THIS_MODULE,
   },
};
```

其 probe 函数 timed_gpio_probe 完成如下工作：

- 分配 num_gpios 个 timed_gpio_data 结构体，每个分别对应需要管理的一个 GPIO
- 对每个 GPIO，调用 hrtimer_init 初始化内核定时器，设置定时器过期的 callback handler 为 gpio_timer_func
- 对每个 GPIO 初始化 timed_gpio_data 结构体其他成员变量，设置 enable 函数为 gpio_enable，get_time 为 gpio_get_time
- 对每个 GPIO，调用 timed_output_dev_register（由 timed_output.c 提供）创建 sysfs 设备

文件，创建 struct device 对象

- 对每个 GPIO，调用 gpio_direction_output 设置其初始输出电平

timed_gpio_remove 函数则做相反的工作。

```
struct timed_gpio_data {
     struct timed_output_dev dev;
     struct hrtimer timer;//关联的定时器
     spinlock_t lock;
     unsigned   gpio;//对应的 GPIO pin
     int max_timeout;//最大允许的 timeout 时间
     u8  active_low;//GPIO 输出的初始电平,同时它还作为个标志使用
};
```

timed_gpio_data 把一个 GPIO 设备与一个 hrtimer 定时器关联。最后一个成员变量 active_low，它的含义是多重的，在 probe 阶段，会根据这个变量设置 GPIO 输出的电平，这时候该变量类似于指定 GPIO 在初始化时候的默认电平。但在后续通过 sysfs 的 enable 函数设置 GPIO 输出电平时候，这个变量则作为一个标志使用，如果 active_low!=0，则将输出电平极性反转，否则不反转。例如：如果调用 gpio_enable(struct timed_output_dev *dev， int value)函数，传进来的 value 参数值是 1，那么如果 active_low==0，则将 GPIO 引脚输出高电平；但是如果 active_low 不等于 0，则输出的电平是低电平。

定时器的 handler 函数 gpio_timer_func 每当设置的 hrtimer 的 timeout 时间到了，自动被内核调用。它调用 gpio_direction_output 让 GPIO 引脚输出相应的电平，具体代码是：

```
static enum hrtimer_restart gpio_timer_func(struct hrtimer *timer)
{
     struct timed_gpio_data *data =
         container_of(timer, struct timed_gpio_data, timer);
     gpio_direction_output(data->gpio, data->active_low ? 1 :  0);
     return HRTIMER_NORESTART;
}
```

可见，该函数让 GPIO 输出的电平取决于 timed_gpio_data->active_low。如果 active_low!=0，则输出高电平，否则输出低电平。

gpio_enable 函数原型为 static void gpio_enable(struct timed_output_dev *dev， int value)，其中第 2 个参数 value 的含义有两个：第一是作为 GPIO 输出的电平，如果 value 不等于 0，则输出高电平，否则输出低电平；其二，value 还被用作重置 hrtimer 定时器的 timeout 时间值。

```
static void gpio_enable(struct timed_output_dev *dev, int value)
{
     struct timed_gpio_data *data = container_of(dev, struct timed_gpio_data,
dev);
     unsigned long flags;
     spin_lock_irqsave(&data->lock, flags);
   /* cancel previous timer and set GPIO according to value */
   hrtimer_cancel(&data->timer);
   gpio_direction_output(data->gpio, data->active_low ? !value :  !!value);
   if (value > 0) {
        if (value > data->max_timeout)
             value = data->max_timeout;
        hrtimer_start(&data->timer,
             ktime_set(value / 1000, (value % 1000) * 1000000),
             HRTIMER_MODE_REL);
   }
```

```
    spin_unlock_irqrestore(&data->lock, flags);
}
```

gpio_enable 函数首先根据参数 value 输出 GPIO 的电平；然后再用 value 参数重置 hrtimer，并重新启动它，这样在 value 指定的 timeout 时间到达后，会再次触发调用 gpio_timer_func。

gpio_get_time 函数调用 hrtimer_get_remaining 得到 timed GPI 关联 hrtimer 的剩余时间。

用户接口

Timed GPIO driver 基于标准 linux 设备模型和 platform driver 框架，利用 sysfs 文件系统暴露 show 和 store 两个标准接口，相应文件为 sys/class/timed_output/<dev_name>/enable，应用可以通过 sysfs 与其交互。