linux 设备模型之 uart 驱动架构分析

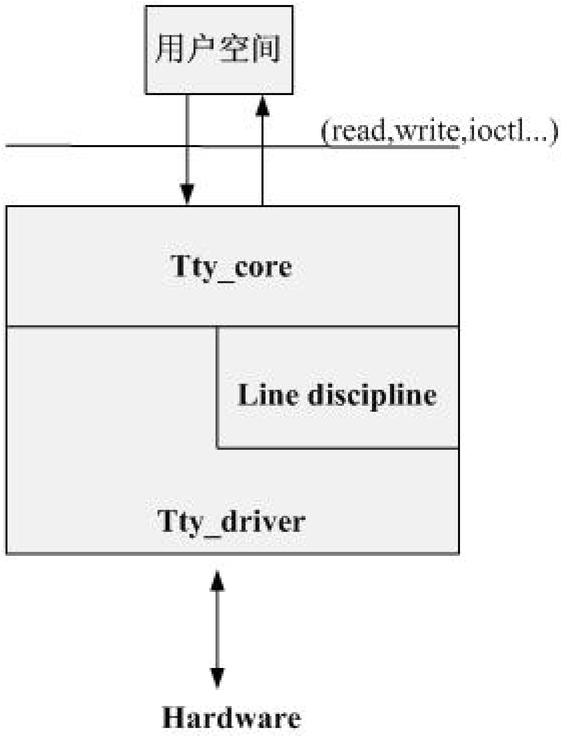
一:前言

接着前面的终端控制台分析,接下来分析 serial 的驱动.在 linux 中,serial 也对应着终端,通常被称为串口终端.在 shell 上,我们看到的/dev/ttyS*就是串口终端所对应的设备节点.

在分析具体的 serial 驱动之前.有必要先分析 uart 驱动架构.uart 是 Universal Asynchronous Receiver and Transmitter 的缩写.翻译成中文即为"通用异步收发器".它是串口设备驱动的封装层.

二:uart 驱动架构概貌

如下图所示:



上图中红色部份标识即为 uart 部份的操作.

从上图可以看到,uart 设备是继 tty_driver 的又一层封装.实际上 uart_driver 就是对应 tty_driver.在它的操作函数中,将操作转入 uart_port.

在写操作的时候,先将数据放入一个叫做 circ_buf 的环形缓存区.然后 uart_port 从缓存区中取数据,将其写入到串口设备中.

当 uart_port 从 serial 设备接收到数据时,会将设备放入对应 line discipline 的缓存区中.

这样.用户在编写串口驱动的时候,只先要注册一个 uart_driver.它的主要作用是定义设备节点号.然后将对设备的各项操作封装在 uart_port.驱动工程师没必要关心上层的流程,只需按硬件规范将 uart_port 中的接口函数完成就可以了.

三:uart 驱动中重要的数据结构及其关联

我们可以自己考虑下,基于上面的架构代码应该要怎么写.首先考虑以下几点:

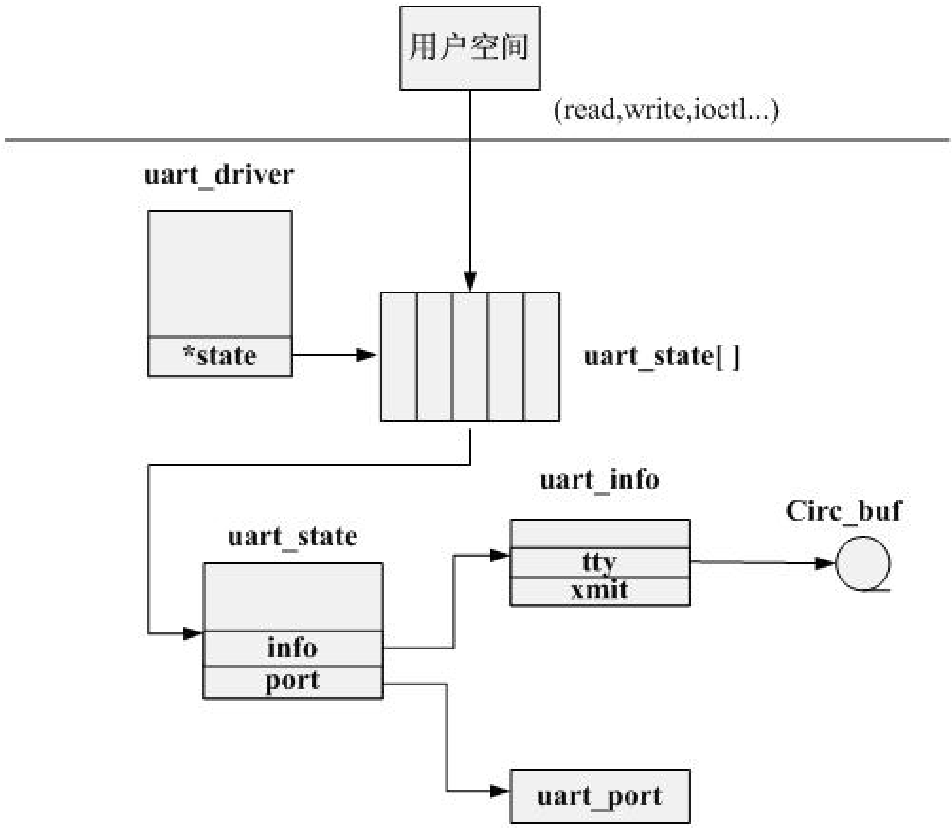
1: 一个 uart_driver 通常会注册一段设备号.即在用户空间会看到 uart_driver 对应有多个设备节点.例如:

/dev/ttyS0 /dev/ttyS1

每个设备节点是对应一个具体硬件的,从上面的架构来看,每个设备文件应该对应一个 uart_port.
也就是说:uart_device 怎么同多个 uart_port 关系起来?怎么去区分操作的是哪一个设备文件?

2:每个 uart_port 对应一个 circ_buf,所以 uart_port 必须要和这个缓存区关系起来

回忆 tty 驱动架构中.tty_driver 有一个叫成员指向一个数组,即 tty->ttys.每个设备文件对应设数组中的一项.而这个数组所代码的数据结构为 tty_struct. 相应的 tty_struct 会将 tty_driver 和 ldisc 关联起来.
那在 uart 驱动中,是否也可用相同的方式来处理呢?
将 uart 驱动常用的数据结构表示如下:



结合上面提出的疑问.可以很清楚的看懂这些结构的设计.

四:uart_driver 的注册操作
Uart_driver 注册对应的函数为: uart_register_driver()代码如下:
```
int uart_register_driver(struct uart_driver *drv)
{
    struct tty_driver *normal = NULL;
    int i, retval;

    BUG_ON(drv->state);

    /*
```

```
	 * Maybe we should be using a slab cache for this, especially if
	 * we have a large number of ports to handle.
	 */
	drv->state = kzalloc(sizeof(struct uart_state) * drv->nr, GFP_KERNEL);
	retval = -ENOMEM;
	if (!drv->state)
		goto out;

	normal  = alloc_tty_driver(drv->nr);
	if (!normal)
		goto out;

	drv->tty_driver = normal;

	normal->owner		= drv->owner;
	normal->driver_name	= drv->driver_name;
	normal->name		= drv->dev_name;
	normal->major		= drv->major;
	normal->minor_start	= drv->minor;
	normal->type		= TTY_DRIVER_TYPE_SERIAL;
	normal->subtype		= SERIAL_TYPE_NORMAL;
	normal->init_termios	= tty_std_termios;
	normal->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
	normal->init_termios.c_ispeed = normal->init_termios.c_ospeed = 9600;
	normal->flags		= TTY_DRIVER_REAL_RAW | TTY_DRIVER_DYNAMIC_DEV;
	normal->driver_state	= drv;
	tty_set_operations(normal, &uart_ops);

	/*
	 * Initialise the UART state(s).
	 */
	for (i = 0; i < drv->nr; i++) {
		struct uart_state *state = drv->state + i;

		state->close_delay	= 500;	/* .5 seconds */
		state->closing_wait	= 30000;	/* 30 seconds */

		mutex_init(&state->mutex);
	}

	retval = tty_register_driver(normal);
out:
	if (retval < 0) {
		put_tty_driver(normal);
```

```
        kfree(drv->state);
    }
    return retval;
}
```

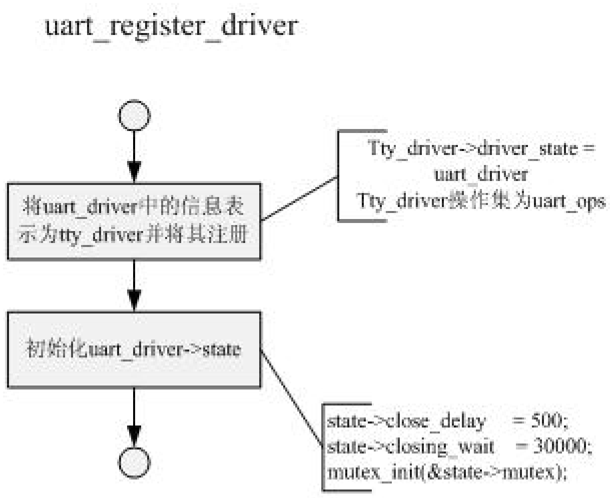
从上面代码可以看出.uart_driver 中很多数据结构其实就是 tty_driver 中的.将数据转换为 tty_driver 之后,注册 tty_driver.然后初始化 uart_driver->state 的存储空间.

这样,就会注册 uart_driver->nr 个设备节点.主设备号为 uart_driver-> major. 开始的次设备号为 uart_driver-> minor.

值得注意的是.在这里将 tty_driver 的操作集统一设为了 uart_ops.其次,在 tty_driver-> driver_state 保存了这个 uart_driver.这样做是为了在用户空间对设备文件的操作时,很容易转到对应的 uart_driver.

另外:tty_driver 的 flags 成员值为: TTY_DRIVER_REAL_RAW | TTY_DRIVER_DYNAMIC_DEV.里面包含有 TTY_DRIVER_DYNAMIC_DEV 标志.结合之前对 tty 的分析.如果包含有这个标志,是不会在初始化的时候去注册 device.也就是说在/dev/下没有动态生成结点(如果是/dev 下静态创建了这个结点就另当别论了^_^).

流程图如下:



五: uart_add_one_port()操作

在前面提到.在对 uart 设备文件过程中.会将操作转换到对应的 port 上,这个 port 跟 uart_driver 是怎么关联起来的呢?这就是 uart_add_ont_port()的主要工作了.

顾名思义,这个函数是在 uart_driver 增加一个 port.代码如下:

```
int uart_add_one_port(struct uart_driver *drv, struct uart_port *port)
{
    struct uart_state *state;
    int ret = 0;
    struct device *tty_dev;

    BUG_ON(in_interrupt());

    if (port->line >= drv->nr)
```

```
	return -EINVAL;

state = drv->state + port->line;

mutex_lock(&port_mutex);
mutex_lock(&state->mutex);
if (state->port) {
	ret = -EINVAL;
	goto out;
}

state->port = port;
state->pm_state = -1;

port->cons = drv->cons;
port->info = state->info;

/*
 * If this port is a console, then the spinlock is already
 * initialised.
 */
if (!(uart_console(port) && (port->cons->flags & CON_ENABLED))) {
	spin_lock_init(&port->lock);
	lockdep_set_class(&port->lock, &port_lock_key);
}

uart_configure_port(drv, state, port);

/*
 * Register the port whether it's detected or not.  This allows
 * setserial to be used to alter this ports parameters.
 */
tty_dev = tty_register_device(drv->tty_driver, port->line, port->dev);
if (likely(!IS_ERR(tty_dev))) {
	device_can_wakeup(tty_dev) = 1;
	device_set_wakeup_enable(tty_dev, 0);
} else
	printk(KERN_ERR "Cannot register tty device on line %d\n",
	       port->line);

/*
 * Ensure UPF_DEAD is not set.
 */
port->flags &= ~UPF_DEAD;
```

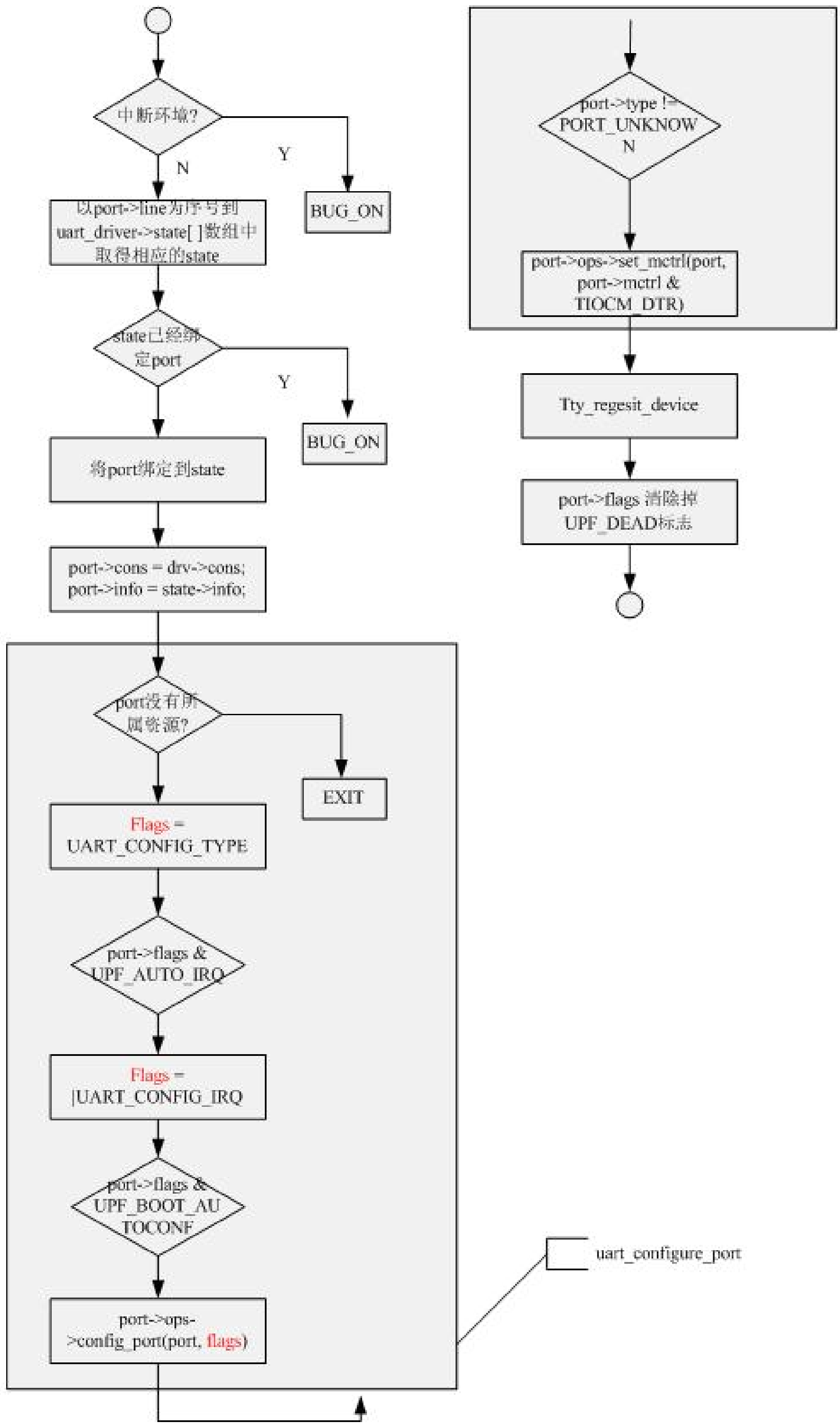
```
 out:
      mutex_unlock(&state->mutex);
      mutex_unlock(&port_mutex);

      return ret;
}
```

首先这个函数不能在中断环境中使用. Uart_port->line 就是对 uart 设备文件序号.它对应的也就是 uart_driver->state 数组中的 uart_port->line 项.

它主要初始化对应 uart_driver->state 项.接着调用 uart_configure_port()进行 port 的自动配置.然后注册 tty_device.如果用户空间运行了 udev 或者已经配置好了 hotplug.就会在/dev 下自动生成设备文件了.

操作流程图如下所示:

uart_add_one_port



六:设备节点的 open 操作

在用户空间执行 open 操作的时候,就会执行 uart_ops->open. Uart_ops 的定义如下:

```
static const struct tty_operations uart_ops = {
     .open          = uart_open,
     .close         = uart_close,
     .write         = uart_write,
     .put_char = uart_put_char,
     .flush_chars   = uart_flush_chars,
     .write_room    = uart_write_room,
     .chars_in_buffer= uart_chars_in_buffer,
     .flush_buffer = uart_flush_buffer,
     .ioctl         = uart_ioctl,
     .throttle = uart_throttle,
     .unthrottle    = uart_unthrottle,
     .send_xchar    = uart_send_xchar,
     .set_termios   = uart_set_termios,
     .stop          = uart_stop,
     .start         = uart_start,
     .hangup        = uart_hangup,
     .break_ctl     = uart_break_ctl,
     .wait_until_sent= uart_wait_until_sent,
#ifdef CONFIG_PROC_FS
     .read_proc     = uart_read_proc,
#endif
     .tiocmget = uart_tiocmget,
     .tiocmset = uart_tiocmset,
};
```

对应 open 的操作接口为 uart_open.代码如下:

```
static int uart_open(struct tty_struct *tty, struct file *filp)
{
     struct uart_driver *drv = (struct uart_driver *)tty->driver->driver_state;
     struct uart_state *state;
     int retval, line = tty->index;

     BUG_ON(!kernel_locked());
     pr_debug("uart_open(%d) called\n", line);

     /*
      * tty->driver->num won't change, so we won't fail here with
      * tty->driver_data set to something non-NULL (and therefore
      * we won't get caught by uart_close()).
      */
     retval = -ENODEV;
     if (line >= tty->driver->num)
```

```
    goto fail;

/*
 * We take the semaphore inside uart_get to guarantee that we won't
 * be re-entered while allocating the info structure, or while we
 * request any IRQs that the driver may need.  This also has the nice
 * side-effect that it delays the action of uart_hangup, so we can
 * guarantee that info->tty will always contain something reasonable.
 */
state = uart_get(drv, line);
if (IS_ERR(state)) {
    retval = PTR_ERR(state);
    goto fail;
}

/*
 * Once we set tty->driver_data here, we are guaranteed that
 * uart_close() will decrement the driver module use count.
 * Any failures from here onwards should not touch the count.
 */
tty->driver_data = state;
tty->low_latency = (state->port->flags & UPF_LOW_LATENCY) ? 1 : 0;
tty->alt_speed = 0;
state->info->tty = tty;

/*
 * If the port is in the middle of closing, bail out now.
 */
if (tty_hung_up_p(filp)) {
    retval = -EAGAIN;
    state->count--;
    mutex_unlock(&state->mutex);
    goto fail;
}

/*
 * Make sure the device is in D0 state.
 */
if (state->count == 1)
    uart_change_pm(state, 0);

/*
 * Start up the serial port.
 */
```

```
    retval = uart_startup(state, 0);

    /*
     * If we succeeded, wait until the port is ready.
     */
    if (retval == 0)
        retval = uart_block_til_ready(filp, state);
    mutex_unlock(&state->mutex);

    /*
     * If this is the first open to succeed, adjust things to suit.
     */
    if (retval == 0 && !(state->info->flags & UIF_NORMAL_ACTIVE)) {
        state->info->flags |= UIF_NORMAL_ACTIVE;

        uart_update_termios(state);
    }

 fail:
    return retval;
}
```

在这里函数里,继续完成操作的设备文件所对应 state 初始化.现在用户空间 open 这个设备了.即要对这个文件进行操作了.那 uart_port 也要开始工作了.即调用 uart_startup()使其进入工作状态.当然,也需要初始化 uart_port 所对应的环形缓冲区 circ_buf.即 state->info-> xmit.

特别要注意,在这里将 tty->driver_data = state;这是因为以后的操作只有 port 相关了,不需要去了解 uart_driver 的相关信息.

跟踪看一下里面调用的两个重要的子函数. uart_get()和 uart_startup().先分析 uart_get().代码如下:

```
static struct uart_state *uart_get(struct uart_driver *drv, int line)
{
    struct uart_state *state;
    int ret = 0;

    state = drv->state + line;
    if (mutex_lock_interruptible(&state->mutex)) {
        ret = -ERESTARTSYS;
        goto err;
    }

    state->count++;
    if (!state->port || state->port->flags & UPF_DEAD) {
        ret = -ENXIO;
        goto err_unlock;
    }
```

```
    if (!state->info) {
        state->info = kzalloc(sizeof(struct uart_info), GFP_KERNEL);
        if (state->info) {
            init_waitqueue_head(&state->info->open_wait);
            init_waitqueue_head(&state->info->delta_msr_wait);

            /*
             * Link the info into the other structures.
             */
            state->port->info = state->info;

            tasklet_init(&state->info->tlet, uart_tasklet_action,
                     (unsigned long)state);
        } else {
            ret = -ENOMEM;
            goto err_unlock;
        }
    }
    return state;

 err_unlock:
    state->count--;
    mutex_unlock(&state->mutex);
 err:
    return ERR_PTR(ret);
}
```

从代码中可以看出.这里注要是操作是初始化state->info.注意port->info就是state->info的一个副本.即port直接通过port->info可以找到它要操作的缓存区.

uart_startup()代码如下:

```
static int uart_startup(struct uart_state *state, int init_hw)
{
    struct uart_info *info = state->info;
    struct uart_port *port = state->port;
    unsigned long page;
    int retval = 0;

    if (info->flags & UIF_INITIALIZED)
        return 0;

    /*
     * Set the TTY IO error marker - we will only clear this
     * once we have successfully opened the port.  Also set
     * up the tty->alt_speed kludge
```

```
 */
set_bit(TTY_IO_ERROR, &info->tty->flags);

if (port->type == PORT_UNKNOWN)
    return 0;

/*
 * Initialise and allocate the transmit and temporary
 * buffer.
 */
if (!info->xmit.buf) {
    page = get_zeroed_page(GFP_KERNEL);
    if (!page)
        return -ENOMEM;

    info->xmit.buf = (unsigned char *) page;
    uart_circ_clear(&info->xmit);
}

retval = port->ops->startup(port);
if (retval == 0) {
    if (init_hw) {
        /*
         * Initialise the hardware port settings.
         */
        uart_change_speed(state, NULL);

        /*
         * Setup the RTS and DTR signals once the
         * port is open and ready to respond.
         */
        if (info->tty->termios->c_cflag & CBAUD)
            uart_set_mctrl(port, TIOCM_RTS | TIOCM_DTR);
    }

    if (info->flags & UIF_CTS_FLOW) {
        spin_lock_irq(&port->lock);
        if (!(port->ops->get_mctrl(port) & TIOCM_CTS))
            info->tty->hw_stopped = 1;
        spin_unlock_irq(&port->lock);
    }

    info->flags |= UIF_INITIALIZED;
```

```
        clear_bit(TTY_IO_ERROR, &info->tty->flags);
    }

    if (retval && capable(CAP_SYS_ADMIN))
        retval = 0;

    return retval;
}
```

在这里,注要完成对环形缓冲,即 info->xmit 的初始化.然后调用 port->ops->startup( )将这个 port 带入到工作状态.其它的是一个可调参数的设置,就不详细讲解了.

七:设备节点的 write 操作

Write 操作对应的操作接口为 uart_write( ).代码如下:

```
static int
uart_write(struct tty_struct *tty, const unsigned char *buf, int count)
{
    struct uart_state *state = tty->driver_data;
    struct uart_port *port;
    struct circ_buf *circ;
    unsigned long flags;
    int c, ret = 0;

    /*
     * This means you called this function _after_ the port was
     * closed.  No cookie for you.
     */
    if (!state || !state->info) {
        WARN_ON(1);
        return -EL3HLT;
    }

    port = state->port;
    circ = &state->info->xmit;

    if (!circ->buf)
        return 0;

    spin_lock_irqsave(&port->lock, flags);
    while (1) {
        c = CIRC_SPACE_TO_END(circ->head, circ->tail, UART_XMIT_SIZE);
        if (count < c)
            c = count;
        if (c <= 0)
            break;
```

```
        memcpy(circ->buf + circ->head, buf, c);
        circ->head = (circ->head + c) & (UART_XMIT_SIZE - 1);
        buf += c;
        count -= c;
        ret += c;
    }
    spin_unlock_irqrestore(&port->lock, flags);

    uart_start(tty);
    return ret;
}
```

Uart_start()代码如下:

```
static void uart_start(struct tty_struct *tty)
{
    struct uart_state *state = tty->driver_data;
    struct uart_port *port = state->port;
    unsigned long flags;

    spin_lock_irqsave(&port->lock, flags);
    __uart_start(tty);
    spin_unlock_irqrestore(&port->lock, flags);
}
static void __uart_start(struct tty_struct *tty)
{
    struct uart_state *state = tty->driver_data;
    struct uart_port *port = state->port;

    if (!uart_circ_empty(&state->info->xmit) && state->info->xmit.buf &&
        !tty->stopped && !tty->hw_stopped)
        port->ops->start_tx(port);
}
```

显然,对于 write 操作而言,它就是将数据 copy 到环形缓存区.然后调用 port->ops->start_tx()将数据写到硬件寄存器.

八:Read 操作

Uart 的 read 操作同 Tty 的 read 操作相同,即都是调用 ldsic->read()读取 read_buf 中的内容.有对这部份内容不太清楚的,参阅<< linux 设备模型之 tty 驱动架构>>.

九:小结

本小节是分析 serial 驱动的基础.在理解了 tty 驱动架构之后,再来理解 uart 驱动架构应该不是很难.随着我们在 linux 设备驱动分析的深入,越来越深刻的体会到,linux 的设备驱动架构很多都是相通的.只要深刻理解了一种驱动架构.举一反三.也就很容易分析出其它架构的驱动了.