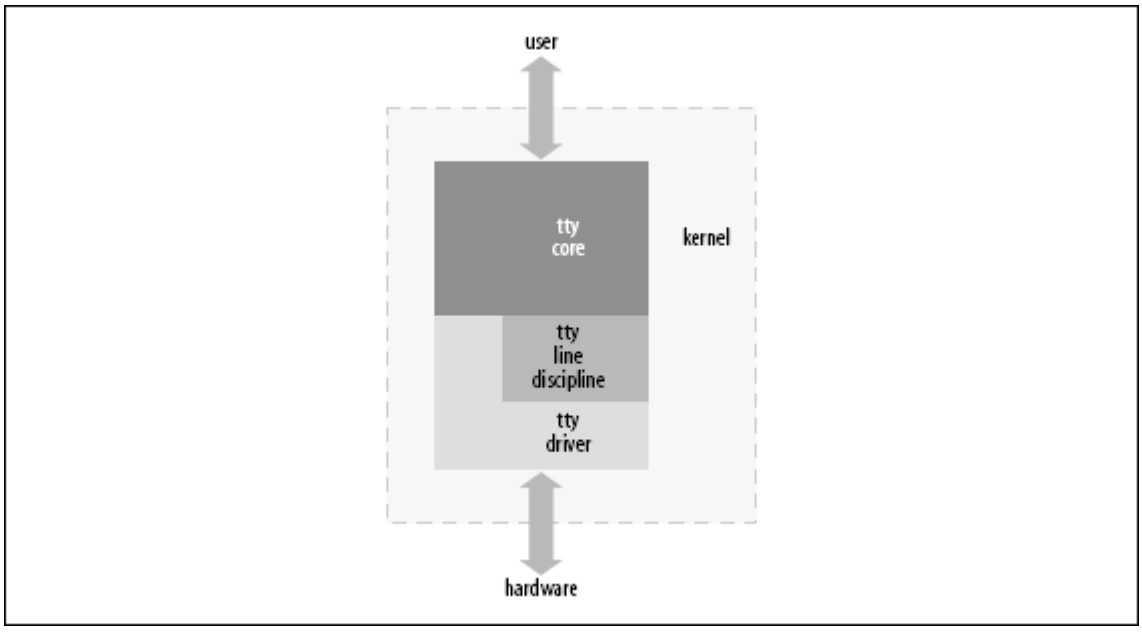
第十八章 TTY驱动

tty 设备的名称来自于 teletypewriter（电传打字机）的非常古老的缩写，最初只与 Unix 机器的物理或虚拟终端连接相关联。随着时间的推移，这个名称也开始意味着任何串行端口样式的设备，因为终端连接也可以通过这样的连接创建。物理 tty 设备的一些例子包括串行端口、USB 到串行端口的转换器，以及需要特殊处理才能正常工作的一些类型的调制解调器（如传统的 Win-Modem 样式设备）。tty 虚拟设备支持用于登录计算机的虚拟控制台，无论是从键盘、通过网络连接，还是通过 xterm 会话。

Linux tty 驱动程序核心位于标准字符驱动程序级别之下，提供了一系列专注于为终端样式设备使用的接口的功能。核心负责控制 tty 设备上数据的流动和数据的格式。这使得 tty 驱动程序可以专注于处理来自硬件的数据和向硬件的数据，而不用担心如何以一致的方式控制与用户空间的交互。为了控制数据的流动，有许多不同的线路规则可以虚拟地“插入”任何 tty 设备。这是通过不同的 tty 线路规则驱动程序完成的。

如图 18-1 所示，tty 核心从用户那里接收要发送到 tty 设备的数据。然后，它将数据传递给 tty 线路规则驱动程序，然后再传递给 tty 驱动程序。tty 驱动程序将数据转换成可以发送到硬件的格式。从 tty 硬件接收的数据通过 tty 驱动程序流回到 tty 线路规则驱动程序，然后流入 tty 核心，用户可以在那里检索它。有时，tty 驱动程序直接与 tty 核心通信，tty 核心直接向 tty 驱动程序发送数据，但通常 tty 线路规则有机会修改在两者之间发送的数据。

tty 驱动程序从未看到 tty 线路规则。驱动程序不能直接与线路规则通信，也不知道它是否存在。驱动程序的工作是以硬件可以理解的方式格式化发送给它的数据，并从硬件接收数据。tty 线路规则的工作是以特定方式格式化从用户或硬件接收的数据。这种格式化通常采取协议转换的形式，如 PPP 或蓝牙。



有三种不同类型的 tty 驱动程序：控制台、串行端口和 pty。控制台和 pty 驱动程序已经被编写出来，可能是这些类型的 tty 驱动程序唯一需要的。这使得任何新的驱动程序使用 tty 核心与用户和系统进行交互，作为串行端口驱动程序。

要确定当前在内核中加载了哪种类型的 tty 驱动程序，以及当前存在哪些 tty 设备，可以查看 `/proc/tty/drivers` 文件。这个文件由当前存在的不同 tty 驱动程序的列表组成，显示驱动程序的名称、默认节点名称、驱动程序的主要编号、驱动程序使用的次要范围，以及 tty 驱动程序的类型。下面是一个这个文件的例子:

```
/dev/tty        /dev/tty        5     0      system:/dev/tty
/dev/console    /dev/console    5     1      system:console
/dev/ptmx       /dev/ptmx       5     2      system
/dev/vc/0       /dev/vc/0       4     0      system:vtmaster
usbserial       /dev/ttyUSB     188   0-254  serial
serial          /dev/ttyS       4     64-67  serial
pty_slave       /dev/pts        136   0-255  pty:slave
pty_master      /dev/ptm        128   0-255  pty:master
pty_slave       /dev/ttyp       3     0-255  pty:slave
pty_master      /dev/pty        2     0-255  pty:master
unknown         /dev/tty        4     1-63   console
```

如果 tty 驱动程序实现了该功能，`/proc/tty/driver/` 目录包含一些 tty 驱动程序的单独文件。默认的串行驱动程序在这个目录中创建一个文件，显示关于硬件的大量串行端口特定信息。如何在这个目录中创建文件的信息将在后面描述。

所有当前在内核中注册并存在的 tty 设备都有自己的子目录在 `/sys/class/tty` 下。在那个子目录中，有一个“dev”文件，包含分配给那个 tty 设备的主要和次要编号。如果驱动程序告诉内核与 tty 设备关联的物理设备和驱动程序的位置，它会创建指向它们的符号链接。

```
/sys/class/tty/
|-- console
| `-- dev
|-- ptmx
| `-- dev
|-- tty
| `-- dev
|-- tty0
| `-- dev
 ...
|-- ttyS1
| `-- dev
|-- ttyS2
| `-- dev
|-- ttyS3
| `-- dev
 ...
|-- ttyUSB0
|  |-- dev
|  |-- device →
../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-
1:1.0/ttyUSB0
|  `-- driver → ../../../bus/usb-serial/drivers/keyspan_4
|-- ttyUSB1
| |-- dev
| |-- device →
../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-
1:1.0/ttyUSB1
| `-- driver → ../../../bus/usb-serial/drivers/keyspan_4
|-- ttyUSB2
| |-- dev
| |-- device →
../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-
1:1.0/ttyUSB2
| `-- driver → ../../../bus/usb-serial/drivers/keyspan_4
`-- ttyUSB3
|-- dev
|-- device →
```

```
../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-
1:1.0/ttyUSB3
`-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
```

18.1. 一个小 TTY 驱动

为了解释 tty 核心是如何工作的，我们创建一个小的 tty 驱动程序，它可以被加载、写入和读取，然后被卸载。任何 tty 驱动程序的主要数据结构是 `struct tty_driver`。它用于在 tty 核心中注册和注销 tty 驱动程序，并在内核头文件 `<linux/tty_driver.h>` 中进行了描述。

以下是一个简单的 tty 驱动程序的示例：

```c
#include <linux/module.h>

#include <linux/tty.h>

#include <linux/tty_driver.h>

#include <linux/tty_flip.h>

static struct tty_driver *my_tty_driver;

static int my_tty_open(struct tty_struct *tty, struct
file *file)

{

    return 0;

}

static void my_tty_close(struct tty_struct *tty, struct
file *file)

{

}

static int my_tty_write(struct tty_struct *tty, const
unsigned char *buf, int count)

{

    return count;

}

static struct tty_operations my_tty_ops = {

    .open = my_tty_open,
```

```
    .close = my_tty_close,
    .write = my_tty_write,
};
static int __init my_tty_init(void)
{
    my_tty_driver = alloc_tty_driver(1);
    if (!my_tty_driver)
        return -ENOMEM;
    my_tty_driver->owner = THIS_MODULE;
    my_tty_driver->driver_name = "my_tty";
    my_tty_driver->name = "my_tty";
    my_tty_driver->major = 240;
    my_tty_driver->type = TTY_DRIVER_TYPE_CONSOLE;
    my_tty_driver->init_termios = tty_std_termios;
    my_tty_driver->init_termios.c_oflag = OPOST | OCRNL;
    tty_set_operations(my_tty_driver, &my_tty_ops);
    if (tty_register_driver(my_tty_driver))
        panic("Couldn't register my_tty driver\n");
    return 0;
```

```
}

static void __exit my_tty_exit(void)

{

    tty_unregister_driver(my_tty_driver);

    put_tty_driver(my_tty_driver);

}

module_init(my_tty_init);

module_exit(my_tty_exit);

MODULE_LICENSE("GPL");

MODULE_DESCRIPTION("Simple TTY Driver");

MODULE_AUTHOR("GitHub Copilot");
```

这个简单的 tty 驱动程序实现了打开、关闭和写入操作。在初始化函数中，它创建一个 tty 驱动程序，设置其操作，并注册到 tty 核心。在退出函数中，它从 tty 核心注销驱动程序并释放资源。

上述变量和函数以及如何使用这个结构将在本章的其余部分中解释。

要将此驱动程序注册到 tty 核心，必须将 `struct tty_driver` 传递给 `tty_register_driver` 函数：

```c
/* register the tty driver */

retval = tty_register_driver(tiny_tty_driver);

if (retval) {

    printk(KERN_ERR "failed to register tiny tty
driver");

    put_tty_driver(tiny_tty_driver);

    return retval;

}
```

当调用 `tty_register_driver` 时，内核为这个 tty 驱动程序可以拥有的整个次要设备范围创建所有不同的 sysfs tty 文件。如果你使用 devfs（本书未涵盖），除非指定了 `TTY_DRIVER_NO_DEVFS` 标志，否则也会创建 devfs 文件。如果你想只为实际存在于系统上的设备调用 `tty_register_device`，那么可以指定这个标志，这样用户总是可以看到内核中存在的设备的最新视图，这是 devfs 用户期望的。

注册自身后，驱动程序通过 `tty_register_device` 函数注册它控制的设备。这个函数有三个参数：

- 设备所属的 `struct tty_driver` 的指针。
- 设备的次要编号。
- 这个 tty 设备绑定到的 `struct device` 的指针。如果 tty 设备没有绑定到任何 `struct device`，这个参数可以设置为 NULL。

我们的驱动程序一次注册所有的 tty 设备，因为它们是虚拟的，没有绑定到任何物理设备：

```c
for (i = 0; i < TINY_TTY_MINORS; ++i)
    tty_register_device(tiny_tty_driver, i, NULL);
```

要在 tty 核心中注销驱动程序，需要通过调用 `tty_unregister_device` 清理所有通过调用 `tty_register_device` 注册的 tty 设备。然后必须通过调用 `tty_unregister_driver` 注销 `struct tty_driver`：

```c
for (i = 0; i < TINY_TTY_MINORS; ++i)
    tty_unregister_device(tiny_tty_driver, i);
tty_unregister_driver(tiny_tty_driver);
```

18.1.1. struct termios

`struct tty_driver` 中的 `init_termios` 变量是一个 `struct termios`。如果在用户初始化端口之前使用端口，此变量用于提供一组合理的线路设置。驱动程序使用一组标准值初始化该变量，这些值是从 `tty_std_termios` 变量复制的。`tty_std_termios` 在 tty 核心中定义为：

```c
struct termios tty_std_termios = {
    .c_iflag = ICRNL | IXON,
    .c_oflag = OPOST | ONLCR,
    .c_cflag = B38400 | CS8 | CREAD | HUPCL,
    .c_lflag = ISIG | ICANON | ECHO | ECHOE | ECHOK |
               ECHOCTL | ECHOKE | IEXTEN,
    .c_cc = INIT_C_CC
};
```

`struct termios` 结构用于保存 tty 设备上特定端口的所有当前线路设置。这些线路设置控制当前的波特率、数据大小、数据流设置和许多其他值。这个结构的不同字段是：

```c
tcflag_t c_iflag;   // 输入模式标志
tcflag_t c_oflag;   // 输出模式标志
tcflag_t c_cflag;   // 控制模式标志
tcflag_t c_lflag;   // 本地模式标志
cc_t c_line;        // 线路纪律类型
cc_t c_cc[NCCS];    // 控制字符数组
```

所有的模式标志都定义为一个大的位字段。模式的不同值及其用途可以在任何 Linux 发行版中可用的 termios manpages 中看到。内核提供了一组有用的宏来获取不同的位。

这些宏在头文件 include/linux/tty.h 中定义。

在 tiny_tty_driver 变量中定义的所有字段都是必需的，以便有一个工作的 tty 驱动程序。 owner 字段是必需的，以防止在 tty 端口打开时卸载 tty 驱动程序。在以前的内核版本中，由 tty 驱动程序自身来处理模块引用计数逻辑。但是内核程序员确定，解决所有可能的竞态条件将是困难的，因此 tty 核心现在为 tty 驱动程序处理所有这些控制。

driver_name 和 name 字段看起来非常相似，但用于不同的目的。
driver_name 变量应设置为短小、描述性强且在所有 tty 驱动程序中唯一的内容，因为它会出现在 /proc/tty/drivers 文件中，向用户描述驱动程序，并出现在当前加载的 tty 驱动程序的 sysfs tty 类目录中。 name 字段用于定义在 /dev 树中分配给此 tty 驱动程序的各个 tty 节点的名称。此字符串用于通过在字符串末尾附加正在使用的 tty 设备的编号来创建 tty 设备。它还用于在 sysfs /sys/class/tty/ 目录中创建设备名称。如果在内核中启用了 devfs，此名称应包括 tty 驱动程序希望放入的任何子目录。例如，如果启用了 devfs，内核中的串行驱动程序将 name 字段设置为 tts/ ，如果没有启用，则设置为 ttyS 。此字符串也显示在 /proc/tty/drivers 文件中。

正如我们提到的， /proc/tty/drivers 文件显示了所有当前注册的 tty 驱动程序。在内核中注册了 tiny_tty 驱动程序并且没有 devfs，这个文件看起来像下面这样：

```
$ cat /proc/tty/drivers

tiny_tty             /dev/ttty     240     0-3 serial

usbserial            /dev/ttyUSB   188   0-254 serial

serial               /dev/ttyS       4  64-107 serial

pty_slave            /dev/pts      136   0-255 pty:slave

pty_master           /dev/ptm      128   0-255 pty:master

pty_slave            /dev/ttyp       3   0-255 pty:slave

pty_master           /dev/pty        2   0-255 pty:master

unknown              /dev/vc/        4    1-63 console

/dev/vc/0            /dev/vc/0       4       0 system:vtma
ster

/dev/ptmx            /dev/ptmx       5       2 system

/dev/console         /dev/console    5       1 system:cons
ole

/dev/tty             /dev/tty        5       0 system:/dev
/tty
```

此外，当 tiny_tty 驱动程序在 tty 核心中注册时，sysfs 目录 `/sys/class/tty` 看起来像下面这样：

```
$ tree /sys/class/tty/tty*

/sys/class/tty/tty0

`-- dev

/sys/class/tty/tty1

`-- dev

/sys/class/tty/tty2

`-- dev

/sys/class/tty/tty3

`-- dev

$ cat /sys/class/tty/tty0/dev

240:0
```

`major` 变量描述了此驱动程序的主要编号是什么。`type` 和 `subtype` 变量声明了此驱动程序是什么类型的 tty 驱动程序。对于我们的示例，我们是一个“正常”类型的串行驱动程序。tty 驱动程序的唯一其他子类型将是“呼叫出”类型。呼叫设备传统上用于控制设备的线路设置。数据将通过一个设备节点发送和接收，任何线路设置更改将发送到另一个设备节点，即呼叫设备。这需要为每个单独的 tty 设备使用两个次要编号。幸运的是，几乎所有驱动程序都在同一设备节点上处理数据和线路设置，新驱动程序很少使用呼叫类型。

`flags` 变量由 tty 驱动程序和 tty 核心使用，以指示驱动程序的当前状态以及它是什么类型的 tty 驱动程序。定义了几个位掩码宏，您必须在测试或操作标志时使用。驱动程序可以设置 `flags` 变量中的三个位：

`TTY_DRIVER_RESET_TERMIOS` 此标志表示当最后一个进程关闭设备时，tty 核心重置 termios 设置。这对于控制台和 pty 驱动程序很有用。例如，假设用户将终端留在一个奇怪的状态。设置此标志后，当用户注销或控制会话的进程被“杀死”时，终端将重置为正常值。

`TTY_DRIVER_REAL_RAW` 此标志表示 tty 驱动程序保证发送奇偶校验或断开字符的通知到线路纪律。这允许线路纪律以更快的方式处理接收到的字符，因为它不必检查从 tty 驱动程序接收的每个字符。由于速度优势，通常为所有 tty 驱动程序设置此值。

`TTY_DRIVER_NO_DEVFS` 此标志表示当调用 `tty_register_driver` 时，tty 核心不为 tty 设备创建任何 devfs 条目。这对于任何动态创建和销毁次设备的驱动程序都很有用。设置此标志的驱动程序的例子包括 USB-to-serial 驱动程序、USB 调制解调器驱动程序、USB 蓝牙 tty 驱动程序以及一些标准串行端口驱动程序。

当 tty 驱动程序稍后想要向 tty 核心注册特定的 tty 设备时，它必须调用 `tty_register_device`，并提供指向 tty 驱动程序的指针，以及已创建的设备的次要编号。如果不这样做，tty 核心仍然将所有调用传递给 tty 驱动程序，但是一些内部 tty 相关的功能可能不存在。这包括新 tty 设备的 `/sbin/hotplug` 通知和 tty 设备的 sysfs 表示。当注册的 tty 设备从机器中移除时，tty 驱动程序必须调用 `tty_unregister_device`。

此变量中的剩余一位由 tty 核心控制，称为 `TTY_DRIVER_INSTALLED`。在驱动程序注册后，tty 核心设置此标志，tty 驱动程序永远不应设置此标志。

18.2. tty_driver 函数指针

最终,驱动被表述为4个函数指针:

18.2.1. open 和 close

`open` 函数在用户调用 tty 驱动程序分配的设备节点上的 `open` 时由 tty 核心调用。tty 核心使用指向分配给此设备的 `tty_struct` 结构的指针和一个文件指针来调用此函数。为了使 tty 驱动程序正常工作，必须设置 `open` 字段；否则，当调用 `open` 时，将返回 `-ENODEV` 给用户。

当调用此 `open` 函数时，预期 tty 驱动程序将在传递给它的 `tty_struct` 变量中保存一些数据，或者在可以基于端口的次要编号引用的静态数组中保存数据。这是必要的，因此当稍后调用 `close`、`write` 和其他函数时，tty 驱动程序知道正在引用哪个设备。

`tiny_tty` 驱动程序在 tty 结构中保存了一个指针，如下面的代码所示：

```c
static int tiny_open(struct tty_struct *tty, struct file
*file)
{
        struct tiny_serial *tiny;
        struct timer_list *timer;
        int index;

        /* initialize the pointer in case something fails
*/
        tty->driver_data = NULL;

        /* get the serial object associated with this tty
pointer */
        index = tty->index;
        tiny = tiny_table[index];
        if (tiny == NULL)
        {

                /* first time accessing this device,
let's create it */
                tiny = kmalloc(sizeof(*tiny),
GFP_KERNEL);
                if (!tiny)

                        return -ENOMEM;
                init_MUTEX(&tiny->sem);
                tiny->open_count = 0;
                tiny->timer = NULL;

                tiny_table[index] = tiny;
        }

        down(&tiny->sem);
        /* save our structure within the tty structure */
        tty->driver_data = tiny;
        tiny->tty = tty;
```

```
        /* increment the usage count */

        down(&tiny->sem);

        tiny->open_count++;

        up(&tiny->sem);



        return 0;

    }
```

这段代码首先检查 `tiny_table` 数组中是否已经有一个 `tiny_serial` 对象与此 tty 设备关联。如果没有，它将创建一个新的 `tiny_serial` 对象并将其保存在 `tiny_table` 数组中。然后，它将 `tiny_serial` 对象的指针保存在 tty 结构的 `driver_data` 字段中，并将 tty 结构的指针保存在 `tiny_serial` 对象的 `tty` 字段中。最后，它增加了 `tiny_serial` 对象的 `open_count` 字段的值，表示设备已被打开。

`tiny_serial` 结构定义如下：

```c
struct tiny_serial {

    struct tty_struct   *tty;        /* pointer to the tty
for this device */

    int          open_count; /* number of times this port
has been opened */

    struct semaphore    sem;         /* locks this
structure */

    struct timer_list   *timer;

};
```

如我们所见，第一次打开端口时，`open_count` 变量在 `open` 调用中初始化为 0。这是一个典型的引用计数器，需要它是因为 tty 驱动程序的 `open` 和 `close` 函数可以为同一设备多次调用，以允许多个进程读取和写入数据。为了正确处理一切，必须保留端口被打开或关闭的次数；驱动程序在使用端口时增加和减少计数。当端口第一次打开时，可以进行任何需要的硬件初始化和内存分配。当端口最后一次关闭时，可以进行任何需要的硬件关闭和内存清理。

`tiny_open` 函数的其余部分显示了如何跟踪设备被打开的次数：

```c
++tiny->open_count;

if (tiny->open_count == 1) {

    /* this is the first time this port is opened */

    /* do any hardware initialization needed here */

}
```

`open` 函数必须返回一个负错误号，如果发生了阻止 `open` 成功的事情，或者返回 0 表示成功。

`close` 函数指针在用户在之前通过调用 `open` 创建的文件句柄上调用 `close` 时由 tty 核心调用。这表明此时应关闭设备。然而，由于 `open` 函数可以被调用多次，`close` 函数也可以被调用多次。因此，此函数应跟踪它被调用的次数，以确定此时是否真的应该关闭硬件。`tiny_tty` 驱动程序使用以下代码来实现这一点：

```c
static void do_close(struct tiny_serial *tiny)
{
        down(&tiny->sem);

        if (!tiny->open_count)
        {
                /* port was never opened */
                goto exit;

        }
        --tiny->open_count;
        if (tiny->open_count <= 0)
        {
                /* The port is being closed by the last
user. */
                /* Do any hardware specific stuff here */

                /* shut down our timer */
                del_timer(tiny->timer);
        }
exit:
        up(&tiny->sem);
}

static void tiny_close(struct tty_struct *tty, struct
file *file)
{
        struct tiny_serial *tiny = tty->driver_data;

        if (tiny)
                do_close(tiny);
}
```

`tiny_close` 函数只是调用 `do_close` 函数来完成关闭设备的实际工作。这样做是为了避免在此处和驱动程序卸载且端口打开时重复关闭逻辑。`close` 函数没有返回值，因为它不应该能够失败。

18.2.2. 数据流

```c
static int tiny_write(struct tty_struct *tty, const
unsigned char *buffer, int count)
{

        struct tiny_serial *tiny = tty->driver_data;
        int i;
        int retval = -EINVAL;
        if (!tiny)
                return -ENODEV;

        down(&tiny->sem);
        if (!tiny->open_count)
                /* port was not opened */
                goto exit;

        /* fake sending the data out a hardware port by
        * writing it to the kernel debug log.
        */
        printk(KERN_DEBUG "%s - ", __FUNCTION__);
        for (i = 0; i < count; ++i)

                printk("%02x ", buffer[i]);
        printk("\n");

exit:
        up(&tiny->sem);
        return retval;

}
```

write 函数在用户有数据要发送到硬件时被调用。首先，tty 核心接收到调用，然后将数据传递给 tty 驱动程序的 write 函数。tty 核心还告诉 tty 驱动程序正在发送的数据的大小。

有时，由于 tty 硬件的速度和缓冲区容量，写入程序请求的所有字符在调用 `write` 函数时可能无法立即发送。`write` 函数应返回能够发送到硬件（或排队稍后发送）的字符数，以便用户程序可以检查是否所有数据确实已写入。在用户空间进行此检查比内核驱动程序坐着等待所有请求的数据能够发送出去要容易得多。如果在 `write` 调用期间发生任何错误，应返回负错误值，而不是已写入的字符数。

`write` 函数可以从中断上下文和用户上下文调用。这很重要，因为当 tty 驱动程序处于中断上下文时，它不应调用可能会睡眠的任何函数。这些包括可能会调用 `schedule` 的任何函数，例如常见的 `copy_from_user`、`kmalloc` 和 `printk` 函数。如果你真的想睡眠，确保首先检查驱动程序是否处于中断上下文，方法是调用 `in_interrupt`。

这个示例的 tiny tty 驱动程序并未连接到任何真实的硬件，所以它的 `write` 函数只是在内核调试日志中记录应该写入什么数据。它使用以下代码来实现这一点：

```c
static int tiny_write(struct tty_struct *tty,
                      const unsigned char *buffer, int
count)
{
    struct tiny_serial *tiny = tty->driver_data;
    int i;
    int retval = -EINVAL;

    if (!tiny)
        return -ENODEV;

    down(&tiny->sem);

    if (!tiny->open_count)
        /* nothing to do */
        goto exit;

    /* fake sending the data out the port */
    printk(KERN_DEBUG "%s - ", __func__);
    for (i = 0; i < count; ++i)
        if (buffer[i] >= 0x20 && buffer[i] < 0x7f)
            printk("%c", buffer[i]);
        else
            printk("\\x%02x", buffer[i]);
    printk("\n");

    retval = count;

exit:
    up(&tiny->sem);
    return retval;
}
```

这段代码首先检查 `tiny_serial` 对象是否存在。如果不存在，它返回 `-ENODEV`。然后，它检查设备是否已打开。如果设备未打开，它什么也不做。然后，它假装将数据发送出端口，实际上是将数据打印到内核调试日志。最后，它返回写入的字符数。

18.2.3. 其他缓冲函数

`tty_driver` 结构中的 `chars_in_buffer` 函数不是必须的，但建议实现。当 tty 核心想知道 tty 驱动程序的写缓冲区中还剩多少字符需要发送出去时，会调用此函数。如果驱动程序可以在将字符发送到硬件之前存储字符，那么它应该实现这个函数，以便 tty 核心能够确定驱动程序中的所有数据是否已经排空。

`tty_driver` 结构中有三个函数回调可以用来刷新驱动程序正在持有的任何剩余数据。这些不是必须实现的，但如果 tty 驱动程序可以在将数据发送到硬件之前缓冲数据，那么建议实现。前两个函数回调被称为 `flush_chars` 和 `wait_until_sent`。当 tty 核心使用 `put_char` 函数回调向 tty 驱动程序发送了一些字符时，会调用这些函数。当 tty 核心希望 tty 驱动程序开始将这些字符发送到硬件（如果还没有开始的话）时，会调用 `flush_chars` 函数回调。这个函数允许在所有数据发送到硬件之前返回。`wait_until_sent` 函数回调的工作方式大致相同；但它必须等到所有字符发送完毕后才返回到 tty 核心，或者等到传入的超时值过期，以先到者为准。tty 驱动程序允许在此函数中睡眠以完成它。如果传递给 `wait_until_sent` 函数回调的超时值设置为 0，那么函数应该等到完成操作后再返回。

剩余数据刷新函数回调是 `flush_buffer`。当 tty 核心要求 tty 驱动程序刷新其写缓冲区中所有仍在内存中的数据时，会调用它。缓冲区中剩余的任何数据都会丢失，不会发送到设备。

18.2.4. 无 read 函数?

虽然 `tty_driver` 结构和 tty 核心没有提供读取函数，也就是说，没有函数回调可以从驱动程序获取数据到 tty 核心。但是，tty 驱动程序负责在接收到硬件的数据时将其发送到 tty 核心。tty 核心会缓冲这些数据，直到用户请求这些数据。由于 tty 核心提供的缓冲逻辑，每个 tty 驱动程序都不必实现自己的缓冲逻辑。当用户希望驱动程序停止和开始发送数据时，tty 核心会通知 tty 驱动程序，但如果内部 tty 缓冲区已满，就不会发出此类通知。

tty 核心在一个名为 `struct tty_flip_buffer` 的结构中缓冲 tty 驱动程序接收的数据。翻转缓冲区是一个包含两个主要数据数组的结构。从 tty 设备接收的数据存储在第一个数组中。当该数组满时，任何等待数据的用户都会收到数据可读的通知。当用户从这个数组读取数据时，任何新的传入数据都会存储在第二个数组中。当该数组完成时，数据再次刷新给用户，驱动程序开始填充第一个数组。本质上，接收的数据从一个缓冲区“翻转”到另一个缓冲区，希望不会溢出两个缓冲区。为了尽量防止数据丢失，tty 驱动程序可以监视传入数组的大小，如果它填满了，就告诉 tty 驱动程序此时刷新缓冲区，而不是等待下一个可用的机会。

struct tty_flip_buffer 结构的细节对 tty 驱动程序来说并不重要，除了一个例外，那就是变量 count。这个变量包含了当前在用于接收数据的缓冲区中剩余的字节数。如果这个值等于 TTY_FLIPBUF_SIZE 的值，那么需要通过调用 tty_flip_buffer_push 将翻转缓冲区刷新给用户。这在以下的代码片段中有所展示：

```c
for (i = 0; i < data_size; ++i) {
    if (tty->flip.count >= TTY_FLIPBUF_SIZE)
        tty_flip_buffer_push(tty);
    tty_insert_flip_char(tty, data[i], TTY_NORMAL);
}
tty_flip_buffer_push(tty);
```

从 tty 驱动程序接收到要发送给用户的字符通过调用 tty_insert_flip_char 添加到翻转缓冲区中。这个函数的第一个参数是应该保存数据的 struct tty_struct，第二个参数是要保存的字符，第三个参数是应该为这个字符设置的任何标志。如果这是一个正常接收的字符，那么标志值应该设置为 TTY_NORMAL。如果这是一个表示接收数据错误的特殊类型的字符，那么应该根据错误设置为 TTY_BREAK、TTY_FRAME、TTY_PARITY 或 TTY_OVERRUN。

为了将数据“推送”给用户，需要调用 tty_flip_buffer_push。如果翻转缓冲区即将溢出，也应该调用这个函数，就像在这个例子中显示的那样。所以，无论何时向翻转缓冲区添加数据，或者当翻转缓冲区满时，tty 驱动程序都必须调用 tty_flip_buffer_push。如果 tty 驱动程序可以以非常高的速率接收数据，那么应该设置 tty->low_latency 标志，这会导致调用 tty_flip_buffer_push 时立即执行。否则，tty_flip_buffer_push 调用会安排自己在不久的将来某个时候将数据从缓冲区推出。

18.3. TTY 线路设置

当用户想要更改 tty 设备的行设置或检索当前的行设置时，他会调用许多不同的 termios 用户空间库函数之一，或者直接在 tty 设备节点上进行 ioctl 调用。tty 核心将

这两种接口转换为许多不同的 tty 驱动程序函数回调和 ioctl 调用。

- 例如，当用户想要更改 tty 设备的行设置时，他可能会调用 termios 库函数 `tcsetattr`。这个函数会被 tty 核心转换为 tty 驱动程序的 `set_termios` 函数回调。同样，当用户想要获取当前的行设置时，他可能会调用 termios 库函数 `tcgetattr`，这个函数会被 tty 核心转换为 tty 驱动程序的 `get_termios` 函数回调。

- 此外，用户也可以直接使用 ioctl 调用来更改或获取 tty 设备的行设置。例如，用户可以使用 `TCSETS` 或 `TCSETSW` ioctl 命令来更改行设置，或者使用 `TCGETS` ioctl 命令来获取当前的行设置。这些 ioctl 调用也会被 tty 核心转换为相应的 tty 驱动程序函数回调。

- 总的来说，tty 核心为用户提供了一个统一的接口来更改或获取 tty 设备的行设置，而无需关心底层的 tty 驱动程序如何实现这些操作。

18.3.1. set_termios 函数

大部分 termios 用户空间函数通过库被转换为对驱动节点的 ioctl 调用。然后，tty 核心将许多不同的 tty ioctl 调用转换为对 tty 驱动程序的单个 `set_termios` 函数调用。`set_termios` 回调需要确定它被要求更改哪些行设置，然后在 tty 设备中进行这些更改。tty 驱动程序必须能够解码 termios 结构中的所有不同设置，并对任何需要的更改做出反应。这是一个复杂的任务，因为所有的行设置都以各种各样的方式打包到 termios 结构中。

`set_termios` 函数应该做的第一件事是确定是否真的需要进行更改。这可以通过以下代码来完成：

```c
unsigned int cflag;

cflag = tty->termios->c_cflag;

/* check that they really want us to change something */

if (old_termios) {

    if ((cflag == old_termios->c_cflag) &&

        (RELEVANT_IFLAG(tty->termios->c_iflag) ==

         RELEVANT_IFLAG(old_termios->c_iflag))) {

        printk(KERN_DEBUG " - nothing to change...\n");

        return;

    }

}
```

RELEVANT_IFLAG 宏定义如下：

```
#define RELEVANT_IFLAG(iflag) ((iflag) & (IGNBRK|BRKINT|IG
NPAR|PARMRK|INPCK))
```

它用于屏蔽 cflags 变量的重要位。然后将此与旧值进行比较，看看它们是否不同。如果不是，那么不需要进行任何更改，所以我们返回。注意，首先检查 old_termios 变量是否指向一个有效的结构，然后再访问它。这是必需的，因为有时这个变量被设置为 NULL。试图访问 NULL 指针的字段会导致内核崩溃。

要查看请求的字节大小，可以使用 CSIZE 位掩码从 cflag 变量中分离出适当的位。如果无法确定大小，通常默认为八个数据位。这可以如下实现：

```c
/* get the byte size */

switch (cflag & CSIZE) {

    case CS5:

        printk(KERN_DEBUG " - data bits = 5\n");

        break;

    case CS6:

        printk(KERN_DEBUG " - data bits = 6\n");

        break;

    case CS7:

        printk(KERN_DEBUG " - data bits = 7\n");

        break;

    default:

    case CS8:

        printk(KERN_DEBUG " - data bits = 8\n");

        break;

}
```

要确定请求的奇偶校验值，可以检查 PARENB 位掩码与 cflag 变量，以告知是否需要设置任何奇偶校验。如果是，可以使用 PARODD 位掩码来确定奇偶校验应该是奇数还是偶数。这的实现如下：

```c
/* determine the parity */
if (cflag & PARENB)
    if (cflag & PARODD)
        printk(KERN_DEBUG " - parity = odd\n");
    else
        printk(KERN_DEBUG " - parity = even\n");
else
    printk(KERN_DEBUG " - parity = none\n");
```

也可以使用 CSTOPB 位掩码从 cflag 变量中确定请求的停止位。这的实现如下：

```c
/* figure out the stop bits requested */
if (cflag & CSTOPB)
    printk(KERN_DEBUG " - stop bits = 2\n");
else
    printk(KERN_DEBUG " - stop bits = 1\n");
```

有两种基本类型的流控制：硬件和软件。要确定用户是否要求硬件流控制，可以检查 CRTSCTS 位掩码与 cflag 变量。这的示例如下：

```c
/* figure out the hardware flow control settings */
if (cflag & CRTSCTS)
    printk(KERN_DEBUG " - RTS/CTS is enabled\n");
else
    printk(KERN_DEBUG " - RTS/CTS is disabled\n");
```

确定不同模式的软件流控制以及不同的停止和开始字符稍微复杂一些：

```c
/* determine software flow control */

/* if we are implementing XON/XOFF, set the start and

 * stop character in the device */

if (I_IXOFF(tty) || I_IXON(tty)) {

    unsigned char stop_char  = STOP_CHAR(tty);

    unsigned char start_char = START_CHAR(tty);

    /* if we are implementing INBOUND XON/XOFF */

    if (I_IXOFF(tty))

        printk(KERN_DEBUG " - INBOUND XON/XOFF is
enabled, "

            "XON = %2x, XOFF = %2x", start_char,
stop_char);

    else

        printk(KERN_DEBUG" - INBOUND XON/XOFF is
disabled");

    /* if we are implementing OUTBOUND XON/XOFF */

    if (I_IXON(tty))

        printk(KERN_DEBUG" - OUTBOUND XON/XOFF is
enabled, "

            "XON = %2x, XOFF = %2x", start_char,
stop_char);

    else
```

```
        printk(KERN_DEBUG" - OUTBOUND XON/XOFF is
disabled");

}
```

最后，需要确定波特率。tty 核心提供了一个函数，`tty_get_baud_rate`，来帮助做这个。该函数返回一个整数，表示特定 tty 设备请求的波特率：

```c
/* get the baud rate wanted */

printk(KERN_DEBUG " - baud rate = %d",
tty_get_baud_rate(tty));
```

现在，tty 驱动程序已经确定了所有不同的行设置，它可以根据这些值正确地设置硬件。

18.3.2. tiocmget 和 tiocmset

在 2.4 和更旧的内核中，曾经有许多 tty ioctl 调用来获取和设置不同的控制线设置。这些由常量 TIOCMGET、TIOCMBIS、TIOCMBIC 和 TIOCMSET 表示。TIOCMGET 用于获取内核的线设置值，从 2.6 内核开始，这个 ioctl 调用已经变成了一个名为 tiocmget 的 tty 驱动程序回调函数。其他三个 ioctls 已经被简化，现在用一个名为 tiocmset 的 tty 驱动程序回调函数表示。

当 tty 核心想知道特定 tty 设备的控制线的当前物理值时，会调用 tty 驱动程序中的 tiocmget 函数。这通常是为了检索串行端口的 DTR 和 RTS 线的值。如果 tty 驱动程序不能直接读取串行端口的 MSR 或 MCR 寄存器，因为硬件不允许这样做，那么应该在本地保存它们的副本。许多 USB-to-serial 驱动程序必须实现这种“阴影”变量。如果保留了这些值的本地副本，这个函数的实现可能是这样的：

```c
static int tiny_tiocmget(struct tty_struct *tty, struct
file *file)

{

    struct tiny_serial *tiny = tty->driver_data;

    unsigned int result = 0;

    unsigned int msr = tiny->msr;

    unsigned int mcr = tiny->mcr;

    result = ((mcr & MCR_DTR)  ? TIOCM_DTR  : 0) |  /*
DTR is set */

             ((mcr & MCR_RTS)  ? TIOCM_RTS  : 0) |  /*
RTS is set */

             ((mcr & MCR_LOOP) ? TIOCM_LOOP : 0) |  /*
LOOP is set */

             ((msr & MSR_CTS)  ? TIOCM_CTS  : 0) |  /*
CTS is set */

             ((msr & MSR_CD)   ? TIOCM_CAR  : 0) |  /*
Carrier detect is set*/

             ((msr & MSR_RI)   ? TIOCM_RI   : 0) |  /*
Ring Indicator is set */

             ((msr & MSR_DSR)  ? TIOCM_DSR  : 0);   /*
DSR is set */

    return result;

}
```

当 tty 核心想要设置特定 tty 设备的控制线的值时，会调用 tty 驱动程序中的 tiocmset 函数。tty 核心通过在两个变量：set 和 clear 中传递它们，告诉 tty 驱动程序要设置和清除哪些值。这些变量包含了应该改变的线设置的位掩码。ioctl 调用从不要求驱动程序同时设置和清除特定的位，所以哪个操作先发生并不重要。这是一个 tty 驱动程序如何实现这个函数的例子：

```c
static int tiny_tiocmset(struct tty_struct *tty, struct
file *file,
                         unsigned int set, unsigned int
clear)
{
    struct tiny_serial *tiny = tty->driver_data;
    unsigned int mcr = tiny->mcr;
    if (set & TIOCM_RTS)
        mcr |= MCR_RTS;
    if (set & TIOCM_DTR)
        mcr |= MCR_RTS;
    if (clear & TIOCM_RTS)
        mcr &= ~MCR_RTS;
    if (clear & TIOCM_DTR)
        mcr &= ~MCR_RTS;
    /* set the new MCR value in the device */
    tiny->mcr = mcr;
    return 0;
}
```

18.4. ioctls 函数

在设备节点上调用 ioctl(2) 时，tty 核心会调用 struct tty_driver 中的 ioctl 函数回调。如果 tty 驱动程序不知道如何处理传递给它的 ioctl 值，它应该返回 -ENOIOCTLCMD，以尝试让 tty 核心实现调用的通用版本。

2.6 内核定义了大约 70 个不同的 tty ioctl，可以发送给 tty 驱动程序。大多数 tty 驱动程序并不处理所有这些 ioctl，而只处理更常见的一小部分。以下是一些更受欢迎的 tty ioctl，它们的含义，以及如何实现它们：

`TIOCSERGETLSR` 获取此 tty 设备的线状态寄存器（LSR）的值。

`TIOCGSERIAL` 获取串行线信息。调用者可以通过这个调用一次性从 tty 设备获取大量的串行线信息。一些程序（如 setserial 和 dip）调用此函数以确保正确设置了波特率，并获取关于 tty 驱动程序控制的设备类型的一般信息。调用者传入一个指向大型 serial_struct 类型结构的指针，tty 驱动程序应该用适当的值填充它。

```c
static int tiny_ioctl(struct tty_struct *tty, struct file
*file, unsigned int cmd, unsigned long arg)
{
        struct tiny_serial *tiny = tty->driver_data;
        if (cmd == TIOCGSERIAL)
        {
                struct serial_struct tmp;
                if (!arg)
                        return -EFAULT;
                memset(&tmp, 0, sizeof(tmp));
                tmp.type  = tiny->serial.type;
                tmp.line  = tiny->serial.line;
                tmp.port  = tiny->serial.port;
                tmp.irq  = tiny->serial.irq;
                tmp.flags  = ASYNC_SKIP_TEST |
ASYNC_AUTO_IRQ;

                tmp.xmit_fifo_size = tiny-
>serial.xmit_fifo_size;
                tmp.baud_base = tiny->serial.baud_base;
                tmp.close_delay = 5*HZ;
                tmp.closing_wait = 30*HZ;
                tmp.custom_divisor = tiny-
>serial.custom_divisor;
                tmp.hub6 = tiny->serial.hub6;
                tmp.io_type = tiny->serial.io_type;
                if (copy_to_user((void __user *)arg,
&tmp, sizeof(tmp)))

                        return -EFAULT;
                return 0;
        }
        return -ENOIOCTLCMD;
}
```

AI生成的示例,比书上的清晰:

```c
static int tiny_ioctl(struct tty_struct *tty,
                      unsigned int cmd, unsigned long
arg)
{
    switch (cmd) {
    case TIOCSERGETLSR:
        return get_lsr_info(tty, (unsigned int __user
*)arg);
    case TIOCGSERIAL:
        return get_serial_info(tty, (struct serial_struct
__user *)arg);
    default:
        return -ENOIOCTLCMD;
    }
}


static int get_lsr_info(struct tty_struct *tty, unsigned
int __user *value)
{
    unsigned int result;
    /* Add code here to get the value of the Line Status
Register */
```

```
    /* ... */

    if (copy_to_user(value, &result, sizeof(int)))

        return -EFAULT;

    return 0;

}



static int get_serial_info(struct tty_struct *tty, struct
serial_struct __user *value)

{

    struct serial_struct info;

    /* Add code here to fill in the fields of 'info' with
the appropriate values */

    /* ... */

    if (copy_to_user(value, &info, sizeof(struct
serial_struct)))

        return -EFAULT;

    return 0;

}
```

TIOCSSERIAL 设置串行线信息。这是 TIOCGSERIAL 的反操作，允许用户一次性设置 tty 设备的串行线状态。这个调用传入一个 struct serial_struct 的指针，其中充满了 tty 设备现在应该设置的数据。如果 tty 驱动程序没有实现这个调用，大多数程序仍然可以正常工作。

`TIOCMIWAIT` 等待 MSR 改变。用户在希望在内核中睡眠，直到 tty 设备的 MSR 寄存器发生某种事情的不寻常情况下请求这个 ioctl。arg 参数包含用户正在等待的事件类型。这通常用于等待状态线改变，表示有更多的数据准备好发送到设备。

实现这个 ioctl 时要小心，不要使用 interruptible_sleep_on 调用，因为它是不安全的（它涉及到许多令人讨厌的竞态条件）。相反，应该使用 wait_queue 来避免这些问题。以下是如何实现这个 ioctl 的示例：

```c
static int tiny_ioctl(struct tty_struct *tty,
                      unsigned int cmd, unsigned long
arg)
{
    switch (cmd) {
    case TIOCSSERIAL:
        return set_serial_info(tty, (struct serial_struct
__user *)arg);
    case TIOCMIWAIT:
        return wait_msr_change(tty, arg);
    default:
        return -ENOIOCTLCMD;
    }
}
static int set_serial_info(struct tty_struct *tty, struct
serial_struct __user *value)
{
    struct serial_struct new_info;
    if (copy_from_user(&new_info, value,
sizeof(new_info)))
        return -EFAULT;
```

```
    /* Add code here to set the tty device based on the
values in 'new_info' */

    /* ... */

    return 0;

}

static int wait_msr_change(struct tty_struct *tty,
unsigned long arg)

{

    struct tiny_serial *tiny = tty->driver_data;

    unsigned long flags;

    int ret;

    spin_lock_irqsave(&tiny->msr_change_lock, flags);

    ret = wait_event_interruptible(tiny->msr_change,

                                   (tiny->msr_prev !=
tiny->msr));

    spin_unlock_irqrestore(&tiny->msr_change_lock,
flags);

    if (!ret)

        ret = put_user(tiny->msr & 0xff, (unsigned int
__user *)arg);

    return ret;

}
```

上面为AI示例，实际的实现将取决于你的硬件和驱动程序的具体需求。
下面是书上示例:

```c
static int tiny_ioctl(struct tty_struct *tty, struct file
*file, unsigned int cmd, unsigned long arg)
{
        struct tiny_serial *tiny = tty->driver_data;
        if (cmd == TIOCMIWAIT)
        {

                DECLARE_WAITQUEUE(wait, current);
                struct async_icount cnow;
                struct async_icount cprev;
                cprev = tiny->icount;
                while (1) {

                        add_wait_queue(&tiny->wait,
&wait);

set_current_state(TASK_INTERRUPTIBLE);
                        schedule();
                        remove_wait_queue(&tiny->wait,
&wait); /* see if a signal woke us up */
                        if (signal_pending(current))
                                return -ERESTARTSYS;
                        cnow = tiny->icount;
                        if (cnow.rng == cprev.rng &&
cnow.dsr == cprev.dsr &&
                                        cnow.dcd ==
cprev.dcd && cnow.cts == cprev.cts)
                                return -EIO; /* no change
=> error */
                        if (((arg & TIOCM_RNG) &&
(cnow.rng != cprev.rng)) || ((arg & TIOCM_DSR) &&
(cnow.dsr != cprev.dsr)) || ((arg & TIOCM_CD) &&
(cnow.dcd != cprev.dcd)) || ((arg & TIOCM_CTS) &&
(cnow.cts != cprev.cts)) ) {
                                return 0;
                        }
                        cprev = cnow;
```

```
                }
        }
        return -ENOIOCTLCMD;

}
```

在 tty 驱动程序的代码中，识别到 MSR 寄存器发生变化的地方，必须调用以下行才能使这段代码正常工作：

```c
wake_up_interruptible(&tp->wait);
```

`TIOCGICOUNT` 获取中断计数。当用户想知道发生了多少串行线中断时，会调用这个。如果驱动程序有一个中断处理程序，它应该定义一个内部结构的计数器来跟踪这些统计信息，并在每次内核运行函数时增加适当的计数器。这个 ioctl 调用将一个指向 serial_icounter_struct 结构的指针传递给内核，应由 tty 驱动程序填充。这个调用通常与前面的 TIOCMIWAIT ioctl 调用一起进行。如果 tty 驱动程序在驱动程序操作时跟踪所有这些中断，实现这个调用的代码可以非常简单：

```c
static int tiny_ioctl(struct tty_struct *tty,
                      unsigned int cmd, unsigned long
arg)
{
    switch (cmd) {
    case TIOCGICOUNT:
        return get_icount(tty, (struct
serial_icounter_struct __user *)arg);
    default:
        return -ENOIOCTLCMD;
    }
}
static int get_icount(struct tty_struct *tty, struct
serial_icounter_struct __user *value)
{
    struct tiny_serial *tiny = tty->driver_data;
    struct serial_icounter_struct icount = tiny->icount;
    if (copy_to_user(value, &icount, sizeof(icount)))
        return -EFAULT;
    return 0;
}
```

上面为AI示例，实际的实现将取决于你的硬件和驱动程序的具体需求。
下面是书上示例:

```c
static int tiny_ioctl(struct tty_struct *tty, struct file
*file, unsigned int cmd, unsigned long arg)
{
        struct tiny_serial *tiny = tty->driver_data;
        if (cmd == TIOCGICOUNT)
        {
                struct async_icount cnow = tiny->icount;
                struct serial_icounter_struct icount;
                icount.cts = cnow.cts;
                icount.dsr = cnow.dsr;
                icount.rng = cnow.rng;
                icount.dcd = cnow.dcd;
                icount.rx = cnow.rx;
                icount.tx = cnow.tx;
                icount.frame = cnow.frame;
                icount.overrun = cnow.overrun;
                icount.parity = cnow.parity;
                icount.brk = cnow.brk;
                icount.buf_overrun = cnow.buf_overrun;
                if (copy_to_user((void __user *)arg,
&icount, sizeof(icount)))

                        return -EFAULT;
                return 0;
        }
        return -ENOIOCTLCMD;
}
```

18.5. TTY 设备的 proc 和 sysfs 处理

tty 核心为任何 tty 驱动程序提供了一种非常简单的方式，可以在 /proc/tty/driver 目录中维护一个文件。如果驱动程序定义了 read_proc 或 write_proc 函数，这个文件就会被创建。然后，对这个文件的任何读或写调用都会发送给驱动程序。这些函数的格式就像标准的 /proc 文件处理函数一样。

作为一个例子，这里是一个简单的实现 read_proc tty 回调的例子，它只是打印出当前注册的端口的数量：

```c
static int tiny_read_proc(char *page, char **start, off_t
off, int count,
                          int *eof, void *data)
{
    struct tiny_serial *tiny;
    off_t begin = 0;
    int length = 0;
    int i;
    length += sprintf(page, "tinyserinfo:1.0
driver:%s\n", DRIVER_VERSION);
    for (i = 0; i < TINY_TTY_MINORS && length <
PAGE_SIZE; ++i) {
        tiny = tiny_table[i];
        if (tiny == NULL)
            continue;
        length += sprintf(page+length, "%d\n", i);
        if ((length + begin) > (off + count))
            goto done;
        if ((length + begin) < off) {
            begin += length;
            length = 0;
```

```
        }
    }
    *eof = 1;
done:
    if (off >= (length + begin))
        return 0;
    *start = page + (off-begin);
    return (count < begin+length-off) ? count : begin +
length-off;
}
```

当注册 tty 驱动程序或创建单个 tty 设备时，tty 核心会处理所有的 sysfs 目录和设备创建，这取决于 struct tty_driver 中的 TTY_DRIVER_NO_DEVFS 标志。单个目录总是包含 dev 文件，这允许用户空间工具确定分配给设备的主要和次要号码。如果在调用 tty_register_device 时传入了指向有效的 struct device 的指针，它还包含一个设备和驱动程序的符号链接。

除了这三个文件，单个 tty 驱动程序无法在此位置创建新的 sysfs 文件。这可能会在未来的内核版本中改变。

18.6. tty_driver 结构的细节

`tty_driver` 结构用于向 tty 核心注册 tty 驱动程序。以下是结构中所有不同字段的列表以及它们如何被 tty 核心使用：

```c
struct module *owner;
```

这个驱动程序的模块所有者。

```c
int magic;
```

这个结构的“魔法”值。应始终设置为 TTY_DRIVER_MAGIC。在 alloc_tty_driver 函数中初始化。

```c
const char *driver_name;
```

驱动程序的名称，用于 /proc/tty 和 sysfs。

```c
const char *name;
```

驱动程序的节点名称。

```c
int name_base;
```

创建设备名称时使用的起始数字。当内核为 tty 驱动程序分配特定的 tty 设备创建字符串表示时使用。

```c
short major;
```

驱动程序的主要编号。

```c
short minor_start;
```

驱动程序的起始次要编号。这通常设置为与 name_base 相同的值。通常，此值设置为 0。

```c
short num;
```

分配给驱动程序的次要编号数量。如果驱动程序使用了整个主要编号范围，此值应设置为 255。此变量在 alloc_tty_driver 函数中初始化。

```c
short type;

short subtype;
```

描述正在向 tty 核心注册的 tty 驱动程序的类型。subtype 的值取决于 type。type 字段可以是：

- TTY_DRIVER_TYPE_SYSTEM
- TTY_DRIVER_TYPE_CONSOLE
- TTY_DRIVER_TYPE_SERIAL
- TTY_DRIVER_TYPE_PTY

```c
struct termios init_termios;
```

设备在创建时的初始 struct termios 值。

```c
int flags;
```

驱动程序标志，如本章前面所述。

```c
struct proc_dir_entry *proc_entry;
```

此驱动程序的 /proc 条目结构。如果驱动程序实现了 write_proc 或 read_proc 函数，它将由 tty 核心创建。此字段不应由 tty 驱动程序本身设置。

```c
struct tty_driver *other;
```

指向 tty 从驱动程序的指针。这只由 pty 驱动程序使用，不应由任何其他 tty 驱动程序使用。

```c
void *driver_state;
```

tty 驱动程序的内部状态。只应由 pty 驱动程序使用。

```c
struct tty_driver *next;

struct tty_driver *prev;
```

链接变量。这些变量由 tty 核心用于将所有不同的 tty 驱动程序链接在一起，任何 tty 驱动程序都不应触摸它们。

18.7. tty_operaions 结构体的细节

`tty_operations` 结构包含了所有可以由 tty 驱动程序设置并由 tty 核心调用的函数回调。目前，这个结构中包含的所有函数指针也都在 `tty_driver` 结构中，但很快就会只用这个结构的一个实例来替换它们。

```c
int (*open)(struct tty_struct * tty, struct file * filp);
```

打开函数。

```c
void (*close)(struct tty_struct * tty, struct file *
filp);
```

关闭函数。

```c
int (*write)(struct tty_struct * tty, const unsigned char
*buf, int count);
```

写入函数。

```c
void (*put_char)(struct tty_struct *tty, unsigned char
ch);
```

单字符写入函数。当需要向设备写入单个字符时，tty 核心会调用此函数。如果 tty 驱动程序没有定义这个函数，当 tty 核心想要发送单个字符时，会调用写入函数。

```c
void (*flush_chars)(struct tty_struct *tty);

void (*wait_until_sent)(struct tty_struct *tty, int
timeout);
```

将数据刷新到硬件的函数。

```c
int (*write_room)(struct tty_struct *tty);
```

表示缓冲区有多少空闲的函数。

```c
int (*chars_in_buffer)(struct tty_struct *tty);
```

表示缓冲区有多少数据的函数。

```c
int (*ioctl)(struct tty_struct *tty, struct file * file,
unsigned int cmd, unsigned long arg);
```

ioctl 函数。当在设备节点上调用 ioctl(2) 时，tty 核心会调用此函数。

```c
void (*set_termios)(struct tty_struct *tty, struct
termios * old);
```

set_termios 函数。当设备的 termios 设置已经改变时，tty 核心会调用此函数。

```c
void (*throttle)(struct tty_struct * tty);

void (*unthrottle)(struct tty_struct * tty);

void (*stop)(struct tty_struct *tty);

void (*start)(struct tty_struct *tty);
```

数据节流函数。这些函数用于帮助控制 tty 核心的输入缓冲区的溢出。当 tty 核心的输入缓冲区快满时，会调用节流函数。tty 驱动程序应该尝试向设备发出信号，表示不应该再向它发送更多的字符。当 tty 核心的输入缓冲区已经清空，现在可以接受更多的数据时，会调用解节流函数。tty 驱动程序应该向设备发出信号，表示可以接收数据。停止和开始函数与节流和解节流函数非常相似，但它们表示 tty 驱动程序应该停止向设备发送数据，然后稍后恢复发送数据。

```c
void (*hangup)(struct tty_struct *tty);
```

挂起函数。当 tty 驱动程序应该挂起 tty 设备时，会调用此函数。此时应该进行任何需要的特殊硬件操作。

```c
void (*break_ctl)(struct tty_struct *tty, int state);
```

线路断开控制函数。当 tty 驱动程序要在 RS-232 端口上打开或关闭线路 BREAK 状态时，会调用此函数。如果 state 设置为 -1，应该打开 BREAK 状态。如果 state 设置为 0，应该关闭 BREAK 状态。如果这个函数由 tty 驱动程序实现，tty 核心将处理 TCSBRK、TCSBRKP、TIOCSBRK 和 TIOCCBRK ioctl。否则，这些 ioctl 将被发送到驱动程序的 ioctl 函数。

```c
void (*flush_buffer)(struct tty_struct *tty);
```

刷新缓冲区并丢弃任何剩余的数据。

```c
void (*set_ldisc)(struct tty_struct *tty);
```

设置线路规程函数。当 tty 核心改变了 tty 驱动程序的线路规程时，会调用此函数。这个函数通常不被使用，驱动程序也不应该定义它。

```c
void (*send_xchar)(struct tty_struct *tty, char ch);
```

发送 X 类型字符函数。此函数用于向 tty 设备发送高优先级的 XON 或 XOFF 字符。要发送的字符在 ch 变量中指定。

```c
int (*read_proc)(char *page, char **start, off_t off, int
count, int *eof, void *data);

int (*write_proc)(struct file *file, const char *buffer,
unsigned long count, void *data);
```

/proc 的读取和写入函数。

```c
int (*tiocmget)(struct tty_struct *tty, struct file
*file);
```

获取特定 tty 设备的当前线路设置。如果从 tty 设备成功检索，应将值返回给调用者。

```c
int (*tiocmset)(struct tty_struct *tty, struct file
*file, unsigned int set, unsigned int clear);
```

设置特定 tty 设备的当前线路设置。set 和 clear 包含了应该被设置或清除的不同线路设置。

18.8. tty_struct 结构体的细节

`tty_struct` 变量由 tty 核心用来保持特定 tty 端口的当前状态。几乎所有的字段都只能由 tty 核心使用，只有少数例外。tty 驱动程序可以使用的字段在这里描述：

```c
unsigned long flags;
```

tty 设备的当前状态。这是一个位字段变量，通过以下宏进行访问：

TTY_THROTTLED 当驱动程序已经调用了节流函数时设置。不应由 tty 驱动程序设置，只能由 tty 核心设置。

TTY_IO_ERROR 当驱动程序不希望从驱动程序读取或写入任何数据时由驱动程序设置。如果用户程序试图这样做，它会从内核接收到一个 -EIO 错误。这通常在设备关闭时设置。

TTY_OTHER_CLOSED 仅由 pty 驱动程序使用，以通知端口已经关闭。

TTY_EXCLUSIVE 由 tty 核心设置，以指示端口处于独占模式，一次只能由一个用户访问。

TTY_DEBUG 在内核中没有使用。

TTY_DO_WRITE_WAKEUP 如果设置了这个，允许调用线路规程的 write_wakeup 函数。这通常在 tty 驱动程序调用 wake_up_interruptible 函数的同时被调用。

TTY_PUSH 仅由默认的 tty 线路规程内部使用。

TTY_CLOSING 由 tty 核心使用，以跟踪端口是否正在关闭。

TTY_DONT_FLIP 由默认的 tty 线路规程使用，通知 tty 核心在设置时不应更改翻转缓冲区。

TTY_HW_COOK_OUT 如果由 tty 驱动程序设置，它会通知线路规程它将“烹饪”发送给它的输出。如果没有设置，线路规程会将驱动程序的输出复制到块中；否则，它必须为每个发送的字节单独评估线路更改。这个标志通常不应由 tty 驱动程序设置。

TTY_HW_COOK_IN 几乎等同于在驱动程序标志变量中设置 TTY_DRIVER_REAL_RAW 标志。这个标志通常不应由 tty 驱动程序设置。

TTY_PTY_LOCK 由 pty 驱动程序使用，以锁定和解锁端口。

TTY_NO_WRITE_SPLIT 如果设置，tty 核心不会将写入到 tty 驱动程序的数据分割成正常大小的块。这个值不应用于防止通过向端口发送大量数据对 tty 端口进行拒绝服务攻击。

```c
struct tty_flip_buffer flip;
```

tty 设备的翻转缓冲区。

```c
struct tty_ldisc ldisc;
```

tty 设备的线路规程。

```c
wait_queue_head_t write_wait;
```

tty 写入函数的等待队列。tty 驱动程序应该唤醒它以信号它可以接收更多的数据。

```c
struct termios *termios;
```

指向 tty 设备当前 termios 设置的指针。

```c
unsigned char stopped:1;
```

指示 tty 设备是否已停止。tty 驱动程序可以设置此值。

```c
unsigned char hw_stopped:1;
```

指示 tty 设备的硬件是否已停止。tty 驱动程序可以设置此值。

```c
unsigned char low_latency:1;
```

指示 tty 设备是否为低延迟设备，能够以非常高的速度接收数据。tty 驱动程序可以设置此值。

```c
unsigned char closing:1;
```

指示 tty 设备是否正在关闭端口。tty 驱动程序可以设置此值。

```c
struct tty_driver driver;
```

当前控制此 tty 设备的 tty_driver 结构。

```c
void *driver_data;
```

tty_driver 可以使用的指针，用于存储本地到 tty 驱动程序的数据。这个变量不会被 tty 核心修改。

18.9. 快速参考

本节提供了对本章介绍的概念的参考. 它还解释了每个 tty 驱动需要包含的头文件的角色. 在 tty_driver 和 tty_device 结构中的成员变量的列表, 但是, 在这里不重复.

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

头文件, 包含 struct tty_driver 的定义和声明一些在这个结构中的不同的标志.

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

头文件, 包含 tty_struct 结构的定义和几个不同的宏定义来易于存取 struct termios 的成员的单个值. 它还含有 tty 驱动核心的函数声明.

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

头文件, 包含几个 tty flip 缓冲内联函数, 使得易于操作 flip 缓冲结构.

```
#include <asm/termios.h>
```

头文件, 包含 struct termio 的定义, 用于内核所建立的特定硬件平台.

```
struct tty_driver *alloc_tty_driver(int lines);
```

函数, 创建一个 struct tty_driver, 可之后传递给 tty_register_driver 和 tty_unregister_driver 函数.

```
void put_tty_driver(struct tty_driver *driver);
```

函数, 清理尚未成功注册到 tty 内核的 struct tty_driver 结构.

```
void tty_set_operations(struct tty_driver *driver, struct
tty_operations *op);
```

函数, 初始化 struct tty_driver 的函数回调. 有必要在 tty_register_driver 可被调用前调用.

```
int tty_register_driver(struct tty_driver *driver);int
tty_unregister_driver(struct tty_driver *driver);
```

函数, 从 tty 核心注册和注销一个 tty 驱动.

```
void tty_register_device(struct tty_driver *driver,
unsigned minor, struct device *device);
void tty_unregister_device(struct tty_driver *driver,
unsigned minor);
```

对 tty 核心注册和注销一个单个 tty 设备的函数.

```
void tty_insert_flip_char(struct tty_struct *tty, unsigned
char ch, char flag);
```

插入字符到 tty 设备的要被用户读的 flip 缓冲的函数.

```
TTY_NORMAL
TTY_BREAK
TTY_FRAME
TTY_PARITY
TTY_OVERRUN
```

flag 参数的不同值, 用在 tty_insert_flip_char 函数.

```
int tty_get_baud_rate(struct tty_struct *tty);
```

函数, 获取当前为特定 tty 设备设置的波特率.

```
void tty_flip_buffer_push(struct tty_struct *tty);
```

函数, 将当前 flip 缓冲中的数据推给用户.

```
tty_std_termios
```

变量, 使用一套通用的缺省线路设置来初始化一个 termios 结构.