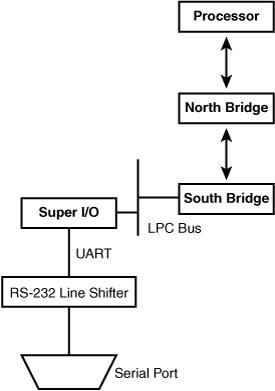
第6章 串行设备驱动

串口是被许多技术和应用广泛使用的基本通信通道。通用异步收发器（Universal Asynchronous Receiver Transmitter，UART）常用来实现串行通信。在PC兼容机硬件上，UART是Super I/O芯片组的一部分，如图6.1所示。

图 6.1. PC串口连接图



尽管RS-232串口是常见的串行硬件，内核的串行子系统还是用通用化的方式组织在一起，以服务不同的用户。你将在如下场合接触到串行子系统：

· 通过RS-232串行链路运行终端会话。

· 通过拨号、小区蜂窝或软件modem连接到Internet。

· 和触摸控制器、智能卡、蓝牙芯片，或红外dongle[MS1] 等使用串行传输方式的设备接口。

· 使用USB到串口的转换器模拟串口。

· 通过RS-485通信，RS-485在RS-232的基础上支持多个节点，传输距离更远，抗噪性能更强。

在本章中，让我们看看内核如何组织串行子系统。我们将利用Linux蜂窝电话作为例子学习底层的UART驱动，利用串行触摸控制器做例子，去发现其高级线路规程的实现细节。

PC机上常见的UART是美国国家半导体有限公司（National Semiconductor）的16550，或者是其他厂家的兼容芯片，因此你将会在代码或文档中看到建议参考“16550类型UART”。8250芯片是16550之前使用的，因此PC上Linux的UART驱动被命名为8250.c。

层次结构

正如你所看到的，串行子系统的用户多种多样。因此促使内核开发者使用如下的构建模块去构造分层次的串行结构：

1. 底层的驱动关注UART或其它底层串行硬件的特征。

2. tty驱动层提供和底层驱动的接口。tty驱动将上层驱动和形形色色的硬件进行了隔离。

3. 线路规程加工用于和tty驱动交换的数据。线路规程勾勒串行层的行为，有助于复用底层的代码来支持不同的技术。

为了帮助用户实现驱动，串行子系统也提供了一些内核API，它们描绘出了子系统中各层的共性。

图6.2展现了不同层次之间的联系。N\_TTY，N\_IRDA，和N\_PPP这3种不同的线路规程分别为串行子系统提供了终端、红外和拨号网络的支持。图6.3显示了串行子系统与内核源文件的映射关系。

图 6.2. 串行子系统层间连接关系图

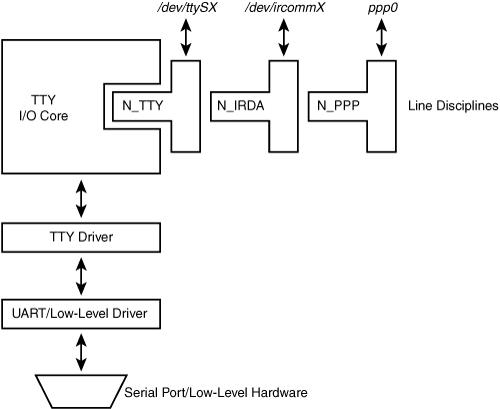
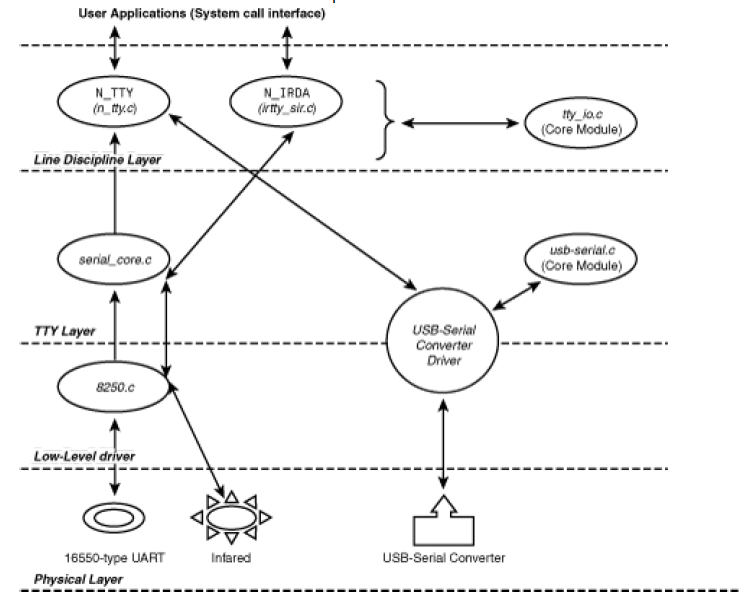
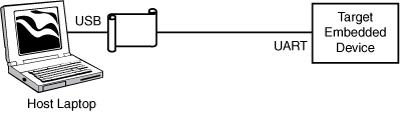


图 6.3. 串行子系统到内核源文件的映射关系



让我们用一个例子来说明层次化的串行结构的好处。假设你在没有串口的笔记本电脑上使用USB串口适配器以获得串口功能。可能的场景就是你将笔记本电脑作为主机端，使用kgdb（kgdb将在第21章“调试设备驱动”中讨论）调试某个嵌入式目标设备的内核，就像图6.4中所展示的。

图6.4. 使用USB到串口转接头



正如在图6.3中所看到的，在你的笔记本电脑上，你首先需要一个合适的USB物理层驱动（和UART驱动相对应的USB部分）。在内核的USB子系统drivers/usb/中提供了此驱动。其次，在USB物理层之上你需要tty驱动。usbserial驱动（drivers/usb/serial/usb-serial.c）为核心层，它实现了基于USB串口转换器的通用tty。usbserial驱动和与设备相关的tty函数一起组成了tty层。其中tty函数由适配器驱动注册（例如，如果你使用Keyspan适配器，其驱动为drivers/usb/serial/keyspan.c）。最后，但不仅仅是这些，你需要N\_TTY线路规程以处理终端I/O。

tty驱动将底层的USB的内部特性与线路规程及高层进行了隔离。实际上，线路规程仍然认为其运行在传统的UART上。其原因是因为tty驱动从USB请求块（USB Request Blocks，将在第11章“通用串行总线”中讨论）中获取数据，并且将数据封装成N\_TTY线路规程期望的格式。层次化的结构简化了实现——所有从线路规程上传的数据能够不作改变的复用。

前述的例子使用了专用的tty驱动和通用的线路规程，相反的用法也很常见。第16章中讨论的红外栈，“无线Linux”，采用的就是通用的tty驱动和被称为N\_IRDA的专用的线路规程。

正如你在图6.2和图6.3中所看到的，虽然UART驱动是字符驱动，但它们并不像我们在前面章节中见到的通常的字符驱动那样，将接口直接暴露给内核系统调用。相反，UART驱动（像下一章中讨论的键盘驱动）提供服务给另一内核层：tty层。I/O系统调用的旅程首先从顶层的线路规程开始，通过tty层，止于UART驱动层。

在本章的剩下章节中，让我们近距离接触串行层的不同驱动组件。我们从底含UART驱动的串行栈的底层开始，行至中间的tty驱动，再向顶层的线路规程驱动进军。

UART驱动

UART驱动围绕三个关键的数据结构展开。这三个数据结构都定义于include/linux/ serial\_core.h中：

1. 特定UART相关的（per-UART）驱动结构，uart\_driver：

struct uart\_driver {

struct module \*owner; /\* 拥有此结构体的模块 \*/

const char \*driver\_name; /\* 名称 \*/

const char \*dev\_name; /\* /dev 节点名称，例如ttyS \*/

/\* ... \*/

int major; /\* 主设备号 \*/

int minor; /\* 次设备号 \*/

/\* ... \*/

struct tty\_driver \*tty\_driver; /\* tty 驱动 \*/

};

结构体中每个域的注释解释了其作用。owner域的作用和前面章节中讨论的file\_operations结构中owner域相同。

2. uart\_port结构。UART驱动拥有的每个端口，都存在uart\_port结构的一个实例。

struct uart\_port {

spinlock\_t lock; /\* 端口锁 \*/

unsigned int iobase; /\* in/out[bwl]\*/

unsigned char \_\_iomem \*membase; /\* read/write[bwl]\*/

unsigned int irq; /\* 中断号 \*/

unsigned int uartclk; /\* base uart clock \*/

unsigned char fifosize; /\* 传输fifo大小\*/

unsigned char x\_char; /\* xon/xoff 流控 \*/

/\* ... \*/

};

3. uart\_ops结构。这个结构是每个UART驱动必须支持的物理硬件上可完成的操作的入口函数的集合。

struct uart\_ops {

uint (\*tx\_empty)(struct uart\_port \*); /\* Is TX FIFO empty? \*/

void (\*set\_mctrl)(struct uart\_port \*,

unsigned int mctrl); /\* Set modem control params \*/

uint (\*get\_mctrl)(struct uart\_port \*); /\* Get modem control params \*/

void (\*stop\_tx)(struct uart\_port \*); /\* Stop xmission \*/

void (\*start\_tx)(struct uart\_port \*); /\* Start xmission \*/

/\* ... \*/

void (\*shutdown)(struct uart\_port \*); /\* Disable the port \*/

void (\*set\_termios)(struct uart\_port \*,

struct termios \*new,

struct termios \*old); /\* Set terminal interface

params \*/

/\* ... \*/

void (\*config\_port)(struct uart\_port \*,

int); /\* Configure UART port \*/

/\* ... \*/

};

UART驱动为了将自身和内核联系起来，必须完成两个重要的步骤：

1. 通过调用uart\_register\_driver(struct uart\_driver \*)向串口核心层注册。

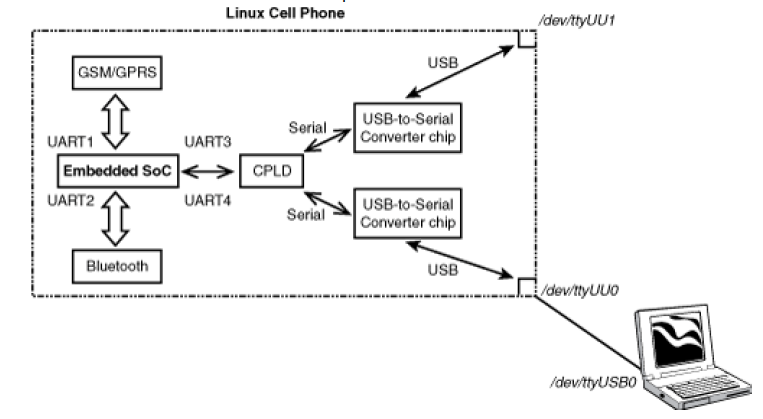
2. 调用uart\_add\_one\_port(struct uart\_driver \*, struct uart\_port \*)注册其支持的每个端口。如果你的串口硬件支持热插拔，探测到设备存在后，从入口点向内核注册。第10章“PCI”的清单10.4中的CardBus Modem驱动，就是串口设备热插拔的例子。需要注意的是一些驱动使用封装的注册函数serial8250\_register\_port(struct uart\_port \*)，在其内部调用了uart\_add\_one\_port()。

这些数据结构和注册函数组成了UART驱动的最基本的共同点。了解了这些结构和例程后，让我们来开发一个UART驱动实例。

设备例子：手机

考虑围绕嵌入式片上系统构建的Linux手机。此片上系统有两个集成的UART，但如图6.5所示，都已经被占用。其中一个用于和调制解调器通信，另一个用于和蓝牙芯片组接口。由于没有空闲的UART用于调试，所以此电话使用了两个USB到串口的转换芯片，一个提供给PC主机作为调试终端，另一个作为备用端口。正如你在本章前面所看到的，USB到串口的转换器使你在PC上通过USB可以连接串口设备，在第11章中，我们将会作更详细的讨论。

图6.5. Linux蜂窝电话上的USB\_UART端口.



两个USB到串口的转换芯片的串口一侧和片上系统通过CPLD（见第18章《嵌入式Linux》中的“CPLD/FPGA”相关章节）连接在一起。CPLD通过创建了两个虚拟的UART（或USB\_UART），它提供了三个寄存器接口以访问每个USB/串口转换器，如表6.1所，这些寄存器为状态寄存器、读数据寄存器和写数据寄存器。为了写一个字符到USB\_UART，程序需要循环检测状态寄存器，当芯片内部的发送FIFO有空间时，状态寄存器位被置位，就可以向写数据寄存器写入字节了。为了读取一个字符，程序将等待直至状态寄存器的相应位显示接收FIFO中有数据，然后从读数据寄存器读取数据。

表 6.1. USB\_UART上的寄存器分布

寄存器名

描述

相对USB\_UART内存基址偏移

UU\_STATUS\_REGISTER

检查发送FIFO是否满或接收FIFO是否空的比特位

0x0

UU\_READ\_DATA\_REGISTER

从USB\_UART读一个字符

0x1

UU\_WRITE\_DATA\_REGISTER

写一个字符至USB\_UART

0x2

在PC一端，使用相应的Linux usbserial驱动（例如，如果在蜂窝电话上使用的是FT232AM芯片，其驱动为drivers/usb/serial/ftdi\_sio.c）以创建并管理对应于USB/串口端口的/dev/ttyUSBX设备节点。你可以运行基于这些设备节点的终端仿真器，例如minicom，以获得控制台或调试终端。在蜂窝电话一端，我们必须为USB\_UART实现UART驱动。此驱动创建并管理负责通信的/dev/ttyUUX节点。

图6.5中的蜂窝电话充当蓝牙设备到GSM网络，以及Internet的智能网关。例如，此电话能够将你的基于蓝牙的血压监控器上的数据传输至你的健康护理提供者在Internet上的服务器上。在和你的基于蓝牙的心率监控器通信过程监测到异常时，它也能够向医生报警。第13章《音频设备驱动》中所使用的蓝牙MP3播放器，以及第16章中使用的喂药棒都是可通过Linux蜂窝电话连接Internet的设备例子。

清单6.1实现了USB\_UART驱动。它是作为一个平台（platform）设备驱动实现的。platform可看作一种伪总线，通常用于将集成进片上系统的的轻量级设备和Linux设备模型连接在一起。platform由如下部分组成：

platform设备。使用和特定结构的安装程序platform\_device\_register()或者其简化版本platform\_device\_register\_simple()添加platform设备。你也可以用platform\_add\_devices()一次添加多个platform设备。定义于include/linux/platform\_device.h的platform\_device结构体代表了一个platform设备：

struct platform\_device {

const char \*name; /\* 设备名称 \*/

u32 id; /\* 此域用于注册一个platform设备的多个实例。

在本例中，两个USB\_UART有不同的ID。 \*/

struct device dev; /\* 包括一个release() 方法和一个平台数据 \*/

/\* ... \*/

};

platform驱动。platform驱动使用platform\_driver\_register()将自身注册进平台中。结构体platform\_driver亦定义于include/linux/platform\_device.h中，代表了platform驱动：

struct platform\_driver {

int (\*probe)(struct platform\_device \*); /\*Probe 方法\*/

int (\*remove)(struct platform\_device \*);/\*Remove 方法\*/

/\* ... \*/

/\* The name field in the following structure should match

the name field in the associated platform\_device

structure \*/

struct device\_driver driver;

};

关于platform设备和platform驱动更详细的文档可参考Documentation/driver-model/platform.txt。为了讨论简单，我们的例子驱动注册了platform设备和platform驱动。

在初始化过程中，首先USB\_UART驱动用uart\_register\_driver()向串口核心层注册自身。初始化成功后，在/proc/tty/drivers中你将会发现以usb\_uart开始的新行。其次，驱动用platform\_device\_register\_simple()注册两个platform设备（每个USB\_UART一个）。正如前面提到的，platform设备通常在电路板引导过程中注册。随后，驱动用platform\_driver\_register()注册platform驱动入口点（probe(), remove(), suspend(), and resume()）。USB\_UARTplatform驱动和前面的platform设备联系在一起，并有一个相应的名字（usb\_uart）。在以上步骤完成后，你将会发现在sysfs下出现两个新目录，每一个和相应的USB\_UART端口相对应：/sys/devices/platform/usb\_uart.0/ 和 /sys/devices/platform/usb\_uart.1/

因为Linux设备层开始检测到和注册的USB\_UART platform设备相匹配的platform驱动，它调用属于platform驱动的probe()入口点[1]（usb\_uart\_probe()），每个USB\_UART一次。probe入口点用uart\_add\_one\_port()添加相应的USB\_UART端口。以上步骤会触发config\_port()入口点（前面讨论的uart\_ops结构的一部分）的调用：声明并映射USB\_UART寄存器空间。如果所有的USB\_UART端口都被成功的添加，串口核心层会发送如下内核消息：

ttyUU0 at MMIO 0xe8000000 (irq = 3) is a USB\_UART

ttyUU1 at MMIO 0xe9000000 (irq = 4) is a USB\_UART

但是，直到某个应用程序打开USB\_UART端口，才会占用中断号。当应用程序关闭USB\_UART端口，中断号被释放。表6.2追溯了驱动代码中声明和释放内存区域与中断号的整个过程。

Table 6.2. Claiming and Freeing Memory and IRQ Resources

Module Insert

usb\_uart\_init()

uart\_register\_driver()

usb\_uart\_probe()

uart\_add\_one\_port()

usb\_uart\_config\_port()

request\_mem\_region()

Module Unload

usb\_uart\_exit()

usb\_unregister\_driver()

usb\_uart\_remove()

uart\_remove\_one\_port()

usb\_uart\_release\_port()

release\_mem\_region()

Open /dev/ttyUUX

usb\_uart\_startup()

request\_irq()

Close /dev/ttyUUX

usb\_uart\_shutdown()

free\_irq()

在发送过程中，驱动收集和UART端口相关的循环缓冲区中的待发送数据。在UART驱动的start\_tx()的入口函数usb\_uart\_start\_tx()可以看到，数据存放于port->info->xmit.buf[port->info->xmit.tail]。

在接收过程中，驱动使用tty\_insert\_flip\_char()和tty\_flip\_buffer\_push()将从USB\_UART收到的数据推出至tty驱动。这些在收中断处理例程usb\_uart\_rxint()中完成的。读者可在读完下一节“TTY驱动”后，再重读此例程。

清单6.1中的注释解释了驱动入口函数和其操作的目的。在uart\_ops中留下了一些入口函数未作实现，以避免让读者陷入过多的细节。

清单6.1. Linux蜂窝电话的USB\_UART驱动

Code View:

#include <linux/console.h>

#include <linux/platform\_device.h>

#include <linux/tty.h>

#include <linux/tty\_flip.h>

#include <linux/serial\_core.h>

#include <linux/serial.h>

#include <asm/irq.h>

#include <asm/io.h>

#define USB\_UART\_MAJOR 200 /\* You've to get this assigned \*/

#define USB\_UART\_MINOR\_START 70 /\* Start minor numbering here \*/

#define USB\_UART\_PORTS 2 /\* The phone has 2 USB\_UARTs \*/

#define PORT\_USB\_UART 30 /\* UART type. Add this to

include/linux/serial\_core.h \*/

/\* Each USB\_UART has a 3-byte register set consisting of

UU\_STATUS\_REGISTER at offset 0, UU\_READ\_DATA\_REGISTER at

offset 1, and UU\_WRITE\_DATA\_REGISTER at offset 2 as shown

in Table 6.1 \*/

#define USB\_UART1\_BASE 0xe8000000 /\* Memory base for USB\_UART1 \*/

#define USB\_UART2\_BASE 0xe9000000 /\* Memory base for USB\_UART2 \*/

#define USB\_UART\_REGISTER\_SPACE 0x3

/\* Semantics of bits in the status register \*/

#define USB\_UART\_TX\_FULL 0x20 /\* TX FIFO is full \*/

#define USB\_UART\_RX\_EMPTY 0x10 /\* TX FIFO is empty \*/

#define USB\_UART\_STATUS 0x0F /\* Parity/frame/overruns? \*/

#define USB\_UART1\_IRQ 3 /\* USB\_UART1 IRQ \*/

#define USB\_UART2\_IRQ 4 /\* USB\_UART2 IRQ \*/

#define USB\_UART\_FIFO\_SIZE 32 /\* FIFO size \*/

#define USB\_UART\_CLK\_FREQ 16000000

static struct uart\_port usb\_uart\_port[]; /\* Defined later on \*/

/\* Write a character to the USB\_UART port \*/

static void

usb\_uart\_putc(struct uart\_port \*port, unsigned char c)

{

/\* Wait until there is space in the TX FIFO of the USB\_UART.

Sense this by looking at the USB\_UART\_TX\_FULL bit in the

status register \*/

while (\_\_raw\_readb(port->membase) & USB\_UART\_TX\_FULL);

/\* Write the character to the data port\*/

\_\_raw\_writeb(c, (port->membase+1));

}

/\* Read a character from the USB\_UART \*/

static unsigned char

usb\_uart\_getc(struct uart\_port \*port)

{

/\* Wait until data is available in the RX\_FIFO \*/

while (\_\_raw\_readb(port->membase) & USB\_UART\_RX\_EMPTY);

/\* Obtain the data \*/

return(\_\_raw\_readb(port->membase+2));

}

/\* Obtain USB\_UART status \*/

static unsigned char

usb\_uart\_status(struct uart\_port \*port)

{

return(\_\_raw\_readb(port->membase) & USB\_UART\_STATUS);

}

/\*

\* Claim the memory region attached to USB\_UART port. Called

\* when the driver adds a USB\_UART port via uart\_add\_one\_port().

\*/

static int

usb\_uart\_request\_port(struct uart\_port \*port)

{

if (!request\_mem\_region(port->mapbase, USB\_UART\_REGISTER\_SPACE,

"usb\_uart")) {

return -EBUSY;

}

return 0;

}

/\* Release the memory region attached to a USB\_UART port.

\* Called when the driver removes a USB\_UART port via

\* uart\_remove\_one\_port().

\*/

static void

usb\_uart\_release\_port(struct uart\_port \*port)

{

release\_mem\_region(port->mapbase, USB\_UART\_REGISTER\_SPACE);

}

/\*

\* Configure USB\_UART. Called when the driver adds a USB\_UART port.

\*/

static void

usb\_uart\_config\_port(struct uart\_port \*port, int flags)

{

if (flags & UART\_CONFIG\_TYPE && usb\_uart\_request\_port(port) == 0)

{

port->type = PORT\_USB\_UART;

}

}

/\* Receive interrupt handler \*/

static irqreturn\_t

usb\_uart\_rxint(int irq, void \*dev\_id)

{

struct uart\_port \*port = (struct uart\_port \*) dev\_id;

struct tty\_struct \*tty = port->info->tty;

unsigned int status, data;

/\* ... \*/

do {

/\* ... \*/

/\* Read data \*/

data = usb\_uart\_getc(port);

/\* Normal, overrun, parity, frame error? \*/

status = usb\_uart\_status(port);

/\* Dispatch to the tty layer \*/

tty\_insert\_flip\_char(tty, data, status);

/\* ... \*/

} while (more\_chars\_to\_be\_read()); /\* More chars \*/

/\* ... \*/

tty\_flip\_buffer\_push(tty);

return IRQ\_HANDLED;

}

/\* Called when an application opens a USB\_UART \*/

static int

usb\_uart\_startup(struct uart\_port \*port)

{

int retval = 0;

/\* ... \*/

/\* Request IRQ \*/

if ((retval = request\_irq(port->irq, usb\_uart\_rxint, 0,

"usb\_uart", (void \*)port))) {

return retval;

}

/\* ... \*/

return retval;

}

/\* Called when an application closes a USB\_UART \*/

static void

usb\_uart\_shutdown(struct uart\_port \*port)

{

/\* ... \*/

/\* Free IRQ \*/

free\_irq(port->irq, port);

/\* Disable interrupts by writing to appropriate

registers \*/

/\* ... \*/

}

/\* Set UART type to USB\_UART \*/

static const char \*

usb\_uart\_type(struct uart\_port \*port)

{

return port->type == PORT\_USB\_UART ? "USB\_UART" : NULL;

}

/\* Start transmitting bytes \*/

static void

usb\_uart\_start\_tx(struct uart\_port \*port)

{

while (1) {

/\* Get the data from the UART circular buffer and

write it to the USB\_UART's WRITE\_DATA register \*/

usb\_uart\_putc(port,

port->info->xmit.buf[port->info->xmit.tail]);

/\* Adjust the tail of the UART buffer \*/

port->info->xmit.tail = (port->info->xmit.tail + 1) &

(UART\_XMIT\_SIZE - 1);

/\* Statistics \*/

port->icount.tx++;

/\* Finish if no more data available in the UART buffer \*/

if (uart\_circ\_empty(&port->info->xmit)) break;

}

/\* ... \*/

}

/\* The UART operations structure \*/

static struct uart\_ops usb\_uart\_ops = {

.start\_tx = usb\_uart\_start\_tx, /\* Start transmitting \*/

.startup = usb\_uart\_startup, /\* App opens USB\_UART \*/

.shutdown = usb\_uart\_shutdown, /\* App closes USB\_UART \*/

.type = usb\_uart\_type, /\* Set UART type \*/

.config\_port = usb\_uart\_config\_port, /\* Configure when driver

adds a USB\_UART port \*/

.request\_port = usb\_uart\_request\_port,/\* Claim resources

associated with a

USB\_UART port \*/

.release\_port = usb\_uart\_release\_port,/\* Release resources

associated with a

USB\_UART port \*/

#if 0 /\* Left unimplemented for the USB\_UART \*/

.tx\_empty = usb\_uart\_tx\_empty, /\* Transmitter busy? \*/

.set\_mctrl = usb\_uart\_set\_mctrl, /\* Set modem control \*/

.get\_mctrl = usb\_uart\_get\_mctrl, /\* Get modem control

.stop\_tx = usb\_uart\_stop\_tx, /\* Stop transmission \*/

.stop\_rx = usb\_uart\_stop\_rx, /\* Stop reception \*/

.enable\_ms = usb\_uart\_enable\_ms, /\* Enable modem status

signals \*/

.set\_termios = usb\_uart\_set\_termios, /\* Set termios \*/

#endif

};

static struct uart\_driver usb\_uart\_reg = {

.owner = THIS\_MODULE, /\* Owner \*/

.driver\_name = "usb\_uart", /\* Driver name \*/

.dev\_name = "ttyUU", /\* Node name \*/

.major = USB\_UART\_MAJOR, /\* Major number \*/

.minor = USB\_UART\_MINOR\_START, /\* Minor number start \*/

.nr = USB\_UART\_PORTS, /\* Number of UART ports \*/

.cons = &usb\_uart\_console, /\* Pointer to the console

structure. Discussed in Chapter

12, "Video Drivers" \*/

};

/\* Called when the platform driver is unregistered \*/

static int

usb\_uart\_remove(struct platform\_device \*dev)

{

platform\_set\_drvdata(dev, NULL);

/\* Remove the USB\_UART port from the serial core \*/

uart\_remove\_one\_port(&usb\_uart\_reg, &usb\_uart\_port[dev->id]);

return 0;

}

/\* Suspend power management event \*/

static int

usb\_uart\_suspend(struct platform\_device \*dev, pm\_message\_t state)

{

uart\_suspend\_port(&usb\_uart\_reg, &usb\_uart\_port[dev->id]);

return 0;

}

/\* Resume after a previous suspend \*/

static int

usb\_uart\_resume(struct platform\_device \*dev)

{

uart\_resume\_port(&usb\_uart\_reg, &usb\_uart\_port[dev->id]);

return 0;

}

/\* Parameters of each supported USB\_UART port \*/

static struct uart\_port usb\_uart\_port[] = {

{

.mapbase = (unsigned int) USB\_UART1\_BASE,

.iotype = UPIO\_MEM, /\* Memory mapped \*/

.irq = USB\_UART1\_IRQ, /\* IRQ \*/

.uartclk = USB\_UART\_CLK\_FREQ, /\* Clock HZ \*/

.fifosize = USB\_UART\_FIFO\_SIZE, /\* Size of the FIFO \*/

.ops = &usb\_uart\_ops, /\* UART operations \*/

.flags = UPF\_BOOT\_AUTOCONF, /\* UART port flag \*/

.line = 0, /\* UART port number \*/

},

{

.mapbase = (unsigned int)USB\_UART2\_BASE,

.iotype = UPIO\_MEM, /\* Memory mapped \*/

.irq = USB\_UART2\_IRQ, /\* IRQ \*/

.uartclk = USB\_UART\_CLK\_FREQ, /\* CLock HZ \*/

.fifosize = USB\_UART\_FIFO\_SIZE, /\* Size of the FIFO \*/

.ops = &usb\_uart\_ops, /\* UART operations \*/

.flags = UPF\_BOOT\_AUTOCONF, /\* UART port flag \*/

.line = 1, /\* UART port number \*/

}

};

/\* Platform driver probe \*/

static int \_\_init

usb\_uart\_probe(struct platform\_device \*dev)

{

/\* ... \*/

/\* Add a USB\_UART port. This function also registers this device

with the tty layer and triggers invocation of the config\_port()

entry point \*/

uart\_add\_one\_port(&usb\_uart\_reg, &usb\_uart\_port[dev->id]);

platform\_set\_drvdata(dev, &usb\_uart\_port[dev->id]);

return 0;

}

struct platform\_device \*usb\_uart\_plat\_device1; /\* Platform device

for USB\_UART 1 \*/

struct platform\_device \*usb\_uart\_plat\_device2; /\* Platform device

for USB\_UART 2 \*/

static struct platform\_driver usb\_uart\_driver = {

.probe = usb\_uart\_probe, /\* Probe method \*/

.remove = \_\_exit\_p(usb\_uart\_remove), /\* Detach method \*/

.suspend = usb\_uart\_suspend, /\* Power suspend \*/

.resume = usb\_uart\_resume, /\* Resume after a suspend \*/

.driver = {

.name = "usb\_uart", /\* Driver name \*/

},

};

/\* Driver Initialization \*/

static int \_\_init

usb\_uart\_init(void)

{

int retval;

/\* Register the USB\_UART driver with the serial core \*/

if ((retval = uart\_register\_driver(&usb\_uart\_reg))) {

return retval;

}

/\* Register platform device for USB\_UART 1. Usually called

during architecture-specific setup \*/

usb\_uart\_plat\_device1 =

platform\_device\_register\_simple("usb\_uart", 0, NULL, 0);

if (IS\_ERR(usb\_uart\_plat\_device1)) {

uart\_unregister\_driver(&usb\_uart\_reg);

return PTR\_ERR(usb\_uart\_plat\_device1);

}

/\* Register platform device for USB\_UART 2. Usually called

during architecture-specific setup \*/

usb\_uart\_plat\_device2 =

platform\_device\_register\_simple("usb\_uart", 1, NULL, 0);

if (IS\_ERR(usb\_uart\_plat\_device2)) {

uart\_unregister\_driver(&usb\_uart\_reg);

platform\_device\_unregister(usb\_uart\_plat\_device1);

return PTR\_ERR(usb\_uart\_plat\_device2);

}

/\* Announce a matching driver for the platform

devices registered above \*/

if ((retval = platform\_driver\_register(&usb\_uart\_driver))) {

uart\_unregister\_driver(&usb\_uart\_reg);

platform\_device\_unregister(usb\_uart\_plat\_device1);

platform\_device\_unregister(usb\_uart\_plat\_device2);

}

return 0;

}

/\* Driver Exit \*/

static void \_\_exit

usb\_uart\_exit(void)

{

/\* The order of unregistration is important. Unregistering the

UART driver before the platform driver will crash the system \*/

/\* Unregister the platform driver \*/

platform\_driver\_unregister(&usb\_uart\_driver);

/\* Unregister the platform devices \*/

platform\_device\_unregister(usb\_uart\_plat\_device1);

platform\_device\_unregister(usb\_uart\_plat\_device2);

/\* Unregister the USB\_UART driver \*/

uart\_unregister\_driver(&usb\_uart\_reg);

}

module\_init(usb\_uart\_init);

module\_exit(usb\_uart\_exit);

RS-485

与RS-232不同，RS-485并不是标准的PC接口，但在嵌入式领域，你可能会碰到计算机和控制系统之间为了可靠通信而使用RS-485的情况。RS-485使用差分信号，因此其传输距离可达数千英尺；而RS-232的传输距离仅为数十英尺。在处理器一端，RS-485接口是半双工的UART操作。因此从发送FIFO发送数据至电缆之前，UART设备驱动需要禁止接收器，使能发送器，这一操作可以通过设置相应GPIO引脚的电平来实现。而为了从电缆上获取数据并传送至接收FIFO，UART驱动需要完成相反的操作。

你必须在串口层中恰当的地方使能/禁止RS-485的接收器/发送器。如果你太早地禁止了发送器，它可能没有足够的时间清空发送FIFO中最后的几个字节数据，这将导致发送数据丢失。相反的，如果你太早地禁止了发送器，你就会阻止此段时间的数据接收，这将导致接收数据丢失。

RS-485支持多节点，因此如果你有多个设备和总线相连，高层的协议必须实现恰当的寻址机制。RS-485不支持使用RTS（Request To Send）和CTS（Clear To Send）的硬流控。

TTY驱动

让我们开始看看tty驱动的核心结构体和注册函数。有三个结构体非常重要：

定义于include/linux/tty.h中的tty\_struct结构体。此结构体包含了和打开的tty相关的所有的状态信息。其结构体成员众多，下面列出了一些重要的成员：

struct tty\_struct {

int magic; /\* Magic marker \*/

struct tty\_driver \*driver; /\* tty驱动指针 \*/

struct tty\_ldisc ldisc; /\* Attached Line

discipline \*/

/\* ... \*/

struct tty\_flip\_buffer flip; /\* Flip Buffer. See

below. \*/

/\* ... \*/

wait\_queue\_head\_t write\_wait; /\* See the section

"Line Disciplines" \*/

wait\_queue\_head\_t read\_wait; /\* See the section

"Line Disciplines" \*/

/\* ... \*/

};

tty\_struct结构体中的tty\_flip\_buffer结构体。这是数据收集和处理机制的中枢：

struct tty\_flip\_buffer {

/\* ... \*/

struct semaphore pty\_sem; /\* Serialize \*/

char \*char\_buf\_ptr; /\* Pointer to the flip

buffer \*/

/\* ... \*/

unsigned char char\_buf[2\*TTY\_FLIPBUF\_SIZE]; /\* The flip

buffer \*/

/\* ... \*/

};

底层的串行驱动将flip缓冲区的一半用于数据收集，线路规程则使用另一半来进行数据处理。数据处理伴随着串行驱动和线路规程所使用的缓冲区指针的移动而持续进行。从drivers/char/tty\_io.c文件的flush\_to\_ldisc()函数中可看到flip的确切行为。

在最近的内核中，tty\_flip\_buffer结构体有些改动，该结构体目前由缓冲区头部（tty\_bufhead）和缓冲区链表（tty\_buffer）组成：

struct tty\_bufhead {

/\* ... \*/

struct semaphore pty\_sem; /\* Serialize \*/

struct tty\_buffer \*head, tail, free; /\* See below \*/

/\* ... \*/

};

struct tty\_buffer {

struct tty\_buffer \*next;

char \*char\_buf\_ptr; /\* Pointer to the flip buffer \*/

/\* ... \*/

unsigned long data[0]; /\* The flip buffer, memory for

which is dynamically

allocated \*/

};

定义于include/linux/tty\_driver.h文件中的tty\_driver结构体。它规定了tty驱动和高层的编程接口：

struct tty\_driver {

int magic; /\* Magic number \*/

/\* ... \*/

int major; /\* Major number \*/

int minor\_start; /\* Start of minor number \*/

/\* ... \*/

/\* Interface routines between a tty driver and higher

layers \*/

int (\*open)(struct tty\_struct \*tty, struct file \*filp);

void (\*close)(struct tty\_struct \*tty, struct file \*filp);

int (\*write)(struct tty\_struct \*tty,

const unsigned char \*buf, int count);

void (\*put\_char)(struct tty\_struct \*tty,

unsigned char ch);

/\* ... \*/

};

像UART驱动一样，tty驱动也需要完成两个步骤以向内核注册自身：

1.

调用tty\_register\_driver(struct tty\_driver \*tty\_d)向tty核心注册自身。

2.

调用

tty\_register\_device(struct tty\_driver \*tty\_d,

unsigned device\_index,

struct device \*device)

注册它支持的每个单独的tty。

本章不会给出tty驱动的实例，在Linux内核中有一些通用的tty驱动：

第16章讨论的蓝牙模拟串口，就是用tty驱动的形式实现的。此驱动（drivers/net/bluetooth/rfcomm/tty.c）在初始化阶段调用tty\_register\_driver()，在处理每个到来的蓝牙连接时调用tty\_register\_device()。

在Linux桌面上，为了使用系统控制台，如果你工作在字符模式下，将需要虚拟终端（virtual terminals，VTs）服务，如果你处在图形模式下，将需要伪终端（pseudo terminals，PTYs）服务。虚拟终端和伪终端都是用tty驱动实现的，分别位于drivers/char/vt.c和drivers/char/pty.c。

传统的UART使用的tty驱动位于drivers/serial/serial\_core.c。

USB到串口转换器的tty驱动位于drivers/usb/serial/usb-serial.c。

线路规程

线路规程提供了一套灵活的机制，使得用户运行不同的应用时，使用相同的串行驱动。底层的物理驱动和tty驱动完成从硬件上收发数据，而线路规程则负责处理这些数据，并在内核空间和用户空间之间进行数据的传递。

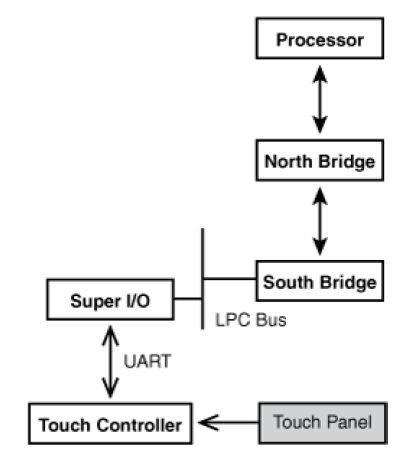
串行子系统支持17种标准的线路规程。当你打开串口时系统会绑定默认的线路规程N\_TTY，它实现了终端I/O处理。N\_TTY负责加工从键盘接收到的字符。根据用户需要，它完成控制字符到“新起一行”的映射，进行小写字符至大写字符的转换，将tab、echo字符传递给关联的虚拟终端。N\_TTY也支持原始编辑模式，此时，它将所有前述的处理都交给用户程序。下一章《输入设备驱动》中的图7.3展示了键盘子系统如何和N\_TTY相关联。前一节“TTY驱动”中列出的tty驱动例子默认就是使用N\_TTY。

线路规程也实现通过串行传输协议的网络接口。PPP（N\_PPP）和SLIP（N\_SLIP）子系统中的线路规程完成将包组帧、分配相应的网络数据结构并将数据传送至相应的网络协议栈的工作。其它的线路规程还包括红外数据（N\_IRDA）和蓝牙主机控制接口（N\_HCI）。

设备例子：触摸控制器

本节通过实现一个简单的串行触摸屏控制器的线路规程，来深入线路规程的内幕。图6.6显示了触摸控制器和嵌入式掌上电脑的连接。由于触摸控制器的有限状态机（FSM）能够很好地描述串行层提供的接口和功用，因此将可根据它实现线路规程。

图 6.6. PC-derivative上触摸控制器连接图



Open与Close

为了创建线路规程，需要定义tty\_ldisc结构体，并向内核注册指定的入口函数集。清单6.2包括了实现了以上功能的触摸控制器实例的部分代码。

清单 6.2. 线路规程操作

Code View:

struct tty\_ldisc n\_touch\_ldisc = {

TTY\_LDISC\_MAGIC, /\* Magic \*/

"n\_tch", /\* 线路规程名 \*/

N\_TCH, /\* 线路规程ID号 \*/

n\_touch\_open, /\* 打开线路规程 \*/

n\_touch\_close, /\* 关闭线路规程 \*/

n\_touch\_flush\_buffer, /\* Flush the line discipline's read

buffer \*/

n\_touch\_chars\_in\_buffer, /\* Get the number of processed characters in

the line discipline's read buffer \*/

n\_touch\_read, /\* Called when data is requested

from user space \*/

n\_touch\_write, /\* Write 方法 \*/

n\_touch\_ioctl, /\* I/O 控制命令 \*/

NULL, /\* We don't have a set\_termios

routine \*/

n\_touch\_poll, /\* 轮询 \*/

n\_touch\_receive\_buf, /\* 由底层的驱动调用，

用于向用户空间传送数据 \*/

n\_touch\_receive\_room, /\* Returns the room left in the line

discipline's read buffer \*/

n\_touch\_write\_wakeup /\* Called when the low-level device

driver is ready to transmit more

data \*/

};

/\* ... \*/

if ((err = tty\_register\_ldisc(N\_TCH, &n\_touch\_ldisc))) {

return err;

}

在清单6.2中，n\_tch是线路规程名，N\_TCH是线路规程的ID号。你需要在include/linux/tty.h中定义其值（此头文件中包括所有线路规程的定义）。在/proc/tty/ldiscs可发现正使用的线路规程。

线路规程从tty的flip缓冲区对应的部分收集、处理数据，然后拷贝处理后的数据至本地读缓冲区。对于N\_TCH，n\_touch\_receive\_room()返回读缓冲区中的剩余内存数，n\_touch\_chars\_in\_buffer()返回读缓冲区中已经处理过的、准备送至用户空间的字符个数。由于N\_TCH是只读设备， n\_touch\_write() 和 n\_touch\_write\_wakeup()将不进行任何操作。n\_touch\_open()用于为线路规程的主要数据结构分配内存，可参照清单6.3。

清单6.3. 打开线路规程

Code View:

/\* Private structure used to implement the Finite State Machine

(FSM) for the touch controller. The controller and the processor

communicate using a specific protocol that the FSM implements \*/

struct n\_touch {

int current\_state; /\* Finite State Machine \*/

spinlock\_t touch\_lock; /\* Spinlock \*/

struct tty\_struct \*tty; /\* Associated tty \*/

/\* Statistics and other housekeeping \*/

/\* ... \*/

} \*n\_tch;

/\* Device open() \*/

static int

n\_touch\_open(struct tty\_struct \*tty)

{

/\* Allocate memory for n\_tch \*/

if (!(n\_tch = kmalloc(sizeof(struct n\_touch), GFP\_KERNEL))) {

return -ENOMEM;

}

memset(n\_tch, 0, sizeof(struct n\_touch));

tty->disc\_data = n\_tch; /\* Other entry points now

have direct access to n\_tch \*/

/\* Allocate the line discipline's local read buffer

used for copying data out of the tty flip buffer \*/

tty->read\_buf = kmalloc(BUFFER\_SIZE, GFP\_KERNEL);

if (!tty->read\_buf) return -ENOMEM;

/\* Clear the read buffer \*/

memset(tty->read\_buf, 0, BUFFER\_SIZE);

/\* Initialize lock \*/

spin\_lock\_init(&ntch->touch\_lock);

/\* Initialize other necessary tty fields.

See drivers/char/n\_tty.c for an example \*/

/\* ... \*/

return 0;

}

当打开连有触摸控制器的串口时，你可能想将N\_TCH而非N\_TTY设置为默认的线路规程，“改变线路规程”一节中介绍的方法可以实现从用户空间改变线路规程的目的。

读数据过程

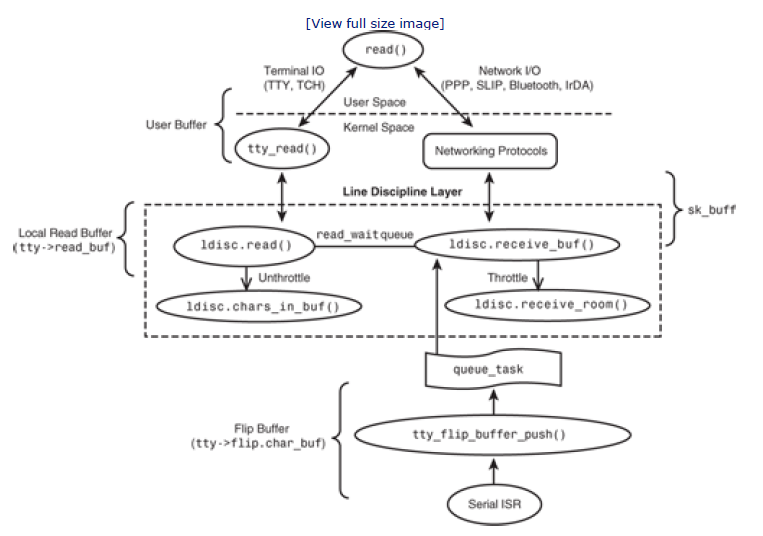
对于中断驱动的设备，读取数据的过程通常由一前一后两个线程组成：

由于发起读数据请求而从用户空间发起的顶层线程；

由中断处理程序（接收来自设备的数据）唤醒的底层线程。

图6.7显示了这两个与读数据流程相关的线程。中断处理程序将receive\_buf()（在我们的例子中是n\_touch\_receive\_buf()）函数当作任务进行排队。通过设置tty->low\_latency可重载这一行为。

图 6.7. 线路规程读数据过程



在触摸控制器的数据手册详细描述了触摸控制器和处理器之间的专用通信协议，而驱动则用前面讨论过的有限状态机来实现此协议。清单6.4中将有限状态机作为receive\_buf()入口点n\_touch\_receive\_buf()的一部分。

清单 6.4. n\_touch\_receive\_buf() 方法

Code View:

static void

n\_touch\_receive\_buf(struct tty\_struct \*tty,

const unsigned char \*cp, char \*fp, int count)

{

/\* Work on the data in the line discipline's half of

the flip buffer pointed to by cp \*/

/\* ... \*/

/\* Implement the Finite State Machine to interpret commands/data

arriving from the touch controller and put the processed data

into the local read buffer \*/

....................................................................................

/\* Datasheet-dependent Code Region \*/

switch (tty->disc\_data->current\_state) {

case RESET:

/\* Issue a reset command to the controller \*/

tty->driver->write(tty, 0, mode\_stream\_command,

sizeof(mode\_stream\_command));

tty->disc\_data->current\_state = STREAM\_DATA;

/\* ... \*/

break;

case STREAM\_DATA:

/\* ... \*/

break;

case PARSING:

/\* ... \*/

tty->disc\_data->current\_state = PARSED;

break;

case PARSED:

/\* ... \*/

}

....................................................................................

if (tty->disc\_data->current\_state == PARSED) {

/\* If you have a parsed packet, copy the collected coordinate

and direction information into the local read buffer \*/

spin\_lock\_irqsave(&tty->disc\_data->touch\_lock, flags);

for (i=0; i < PACKET\_SIZE; i++) {

tty->disc\_data->read\_buf[tty->disc\_data->read\_head] =

tty->disc\_data->current\_pkt[i];

tty->disc\_data->read\_head =

(tty->disc\_data->read\_head + 1) & (BUFFER\_SIZE - 1);

tty->disc\_data->read\_cnt++;

}

spin\_lock\_irqrestore(&tty->disc\_data->touch\_lock, flags);

/\* ... \*/ /\* See Listing 6.5 \*/

}

}

n\_touch\_receive\_buf() 处理从串行驱动来的数据。它和触摸控制器进行一系列命令/响应的交互，将接收的坐标和压下/释放信息放入线路规程读缓冲区。对读缓冲区的访问必须借助自旋锁的加锁能力来依次进行，如图6.7所示，该自旋锁被ldisc.receive\_buf() 和 ldisc.read() 线程（在我们的例子中分别是n\_touch\_receive\_buf() 和 n\_touch\_read()）同时使用。如清单6.4，n\_touch\_receive\_buf()通过直接调用串行驱动的write()入口函数将命令分发给触摸控制器。

n\_touch\_receive\_buf()需要做如下操作：

如果没有数据可获得，图6.7中的顶层的read()线程会将调用进程置为休眠状态。因此n\_touch\_receive\_buf()必须将其唤醒，使其读取刚处理过的数据。

如果线路规程耗尽了读缓冲空间，n\_touch\_receive\_buf()必须要求串行驱动中止从设备接收数据。当它将数据搬移至用户空间并释放读缓冲区中的内存后，ldisc.read()负责重新开启数据接收。串行驱动利用软件或硬件流控机制完成数据接收的中止和重启。

清单6.5实现了上面的操作。

清单 6.5. 唤醒读线程和中止串行驱动

/\* n\_touch\_receive\_buf() continued.. \*/

/\* Wake up any threads waiting for data \*/

if (waitqueue\_active(&tty->read\_wait) &&

(tty->read\_cnt >= tty->minimum\_to\_wake))

wake\_up\_interruptible(&tty->read\_wait);

}

/\* If we are running out of buffer space, request the

serial driver to throttle incoming data \*/

if (n\_touch\_receive\_room(tty) < TOUCH\_THROTTLE\_THRESHOLD) {

tty->driver.throttle(tty);

}

/\* ... \*/

等待队列tty->read\_wait用于在ldisc.read()和ldisc.receive\_buf()线程之间实现同步。当ldisc.read()未发现可读的数据时，将调用进程加入等待队列，当有数据可读时，ldisc.receive\_buf()唤醒ldisc.read()线程。因此，n\_touch\_read()完成如下操作：

· 当仍然无可读数据时，将调用进程放入read\_wait队列中使其休眠。当数据到来时由n\_touch\_receive\_buf()唤醒此进程。

· 若数据可获得，从本地读缓冲区（tty->read\_buf[tty->read\_tail]）中收集数据，并分发至用户空间。

· 若串行驱动被中止，并且在读操作后读缓冲区中又有了足够的可用空间，请求串行驱动重启。

网络线路规程通常分配sk\_buff（第15章“网络接口卡”中将讨论到的基本的Linux网络数据结构），并用它作读缓冲区。由于网络线路规程的receive\_buf()将接收的拷贝至sk\_buff并将其直接传送至相应的协议栈，因此它没有read()函数。

写数据过程

线路规程的write()入口函数需要完成一些数据传送至底层驱动之前必要的后处理工作。

如果底层驱动不能接受线路规程提供的所有数据，线路规程会将请求发送数据的线程置于休眠状态。当驱动准备好接受更多的数据时，驱动的中断处理例程将线路规程唤醒。为了达成此目的，驱动调用由线路规程注册的write\_wakeup()方法。类似于前面章节中讨论过的read\_wait实现同步，此处通过等待队列tty->write\_wait来实现相应的同步。

很多网络线路规程没有write()方法。协议实现时直接将数据帧向下传送给串行设备驱动。然而，这些线路规程通常仍然有write\_wakeup()入口点，以响应串行驱动的传输更多数据的请求。

因为触摸控制器是只读设备，N\_TCH也没有write()方法。正如在清单6.4中所见，当需要发送命令帧给控制器时，接收路径中的例程直接和底层的UART驱动交互。

I/O 控制

像第5章《字符设备驱动》中讨论的那样，用户程序可以通过调用ioctl()向设备发送命令。当一个应用程序打开一个串口设备时，它通常可向其发出三类ioctl：

· 串行驱动设备支持的命令，例如设置modem信息的TIOCMSET。

· tty驱动支持的命令，例如改变关联的线路规程的TIOCSETD。

· 关联的线路规程支持的命令，例如在N\_TCH例子中，复位触摸控制器的命令。

N\_TCH中实现的ioctl()基本是标准的。支持的命令依赖于触摸控制器的数据手册中描述的协议。

其他操作

另一个线路规程操作是flush\_buffer()，用于清理读缓冲区中未处理的数据。当线路规程关闭时，也会调用flush\_buffer()。它唤醒所有等待数据的读线程，其操作如下：

if (tty->link->packet){

wake\_up\_interruptible(&tty->disc\_data->read\_wait);

}

还有一个入口函数（N\_TCH不支持）为set\_termios()。N\_TTY线路规程支持set\_termios()接口，用于进行和线路规程数据处理相关的配置。例如你可以用set\_termios()将线路规程置为原始模式或加工模式。一些触摸控制器特定的配置（如改变波特率、校验和停止位）由底层设备驱动的set\_termios()方法提供。

其他的入口函数例如poll()都是一些标准入口函数，如果需要可以参阅第五章。

你可以将线路规程编译进内核或编译为模块动态加载。如果你选择编译为模块，就必须提供模块初始化或释放需要调用的函数。前者通常使用和init()类似的方法，后者需要清空私有的数据结构，并注销线路规程。注销线路规程的操作为：

tty\_unregister\_ldisc(N\_TCH);

驱动串行触摸控制器更简单的途径是利用内核输入子系统提供的服务和内嵌的serport线路规程。在后续章节中我们将看到其技术细节。

修改线路规程

当用户空间程序打开和触摸控制器相连的串口时，N\_TCH将被绑定到底层的串行驱动。但有时，用户空间的应用程序可能需要给设备绑定其他的线路规程。例如，你可能想编写程序清空触摸控制器接收的所有原始数据而不处理它。清单6.6打开触摸控制器，并修改线路规程为N\_TTY并清空所有接收的数据。

清单 6.6. 从用户空间修改线路规程

fd = open("/dev/ttySX", O\_RDONLY | O\_NOCTTY);

/\* At this point, N\_TCH is attached to /dev/ttySX, the serial port used

by the touch controller. Switch to N\_TTY \*/

ldisc = N\_TTY;

ioctl(fd, TIOCSETD, &ldisc);

/\* Set termios to raw mode and dump the data coming in \*/

/\* ... \*/

ioctl()的TIOCSETD命令关闭当前的线路规程并打开新选定的线路规程。

查看源代码

串行核心位于drivers/serial/，tty实现和底层的驱动分散在内核源码树中。例如，驱动文件可参考图6.3，位于四个不同的目录中：drivers/serial/，drivers/char/，drivers/usb/serial/和 drivers/net/irda/。drivers/serial/目录包含UART驱动，而在2.4版本的内核中无此目录，UART相关的代码过去通常分散在drivers/char/ 和 arch/your-arch/目录中。现在的代码划分较之以前更有逻辑性，因为UART驱动不是串行层的唯一访问者，USB到串口转换器以及红外dongle等设备也需要和串行核心交互。

drivers/serial/imx.c是一个实际的、底层的UART驱动，它驱动的是飞思卡尔（Freescale）i.MX系列嵌入式控制器的UART。

Linux支持的线路规程列表可以参照include/linux/tty.h文件。如果想了解网络线路规程，可以阅读相应的源文件：PPP（drivers/net/ppp\_async.c），蓝牙（drivers/bluetooth/hci\_ldisc.c）， 红外（drivers/net/irda/irtty-sir.c）和SLIP（drivers/net/slip.c）。

表6.3包括本章中用到的主要数据结构的概述和它们在源码树中被定义的位置。表6.4列出了本章用到的主要内核编程接口和其定义位置。

表 6.3. 数据结构概述

数据结构

位置

描述

uart\_driver

include/linux/serial\_core.h

代表底层的UART驱动。

uart\_port

include/linux/serial\_core.h

代表一个UART端口。

uart\_ops

include/linux/serial\_core.h

UART驱动支持的入口函数。

platform\_device

include/linux/platform\_device.h

代表platform设备。

platform\_driver

include/linux/platform\_device.h

代表一个platform驱动。

tty\_struct

include/linux/tty.h

tty的状态信息。

tty\_bufhead, tty\_buffer

include/linux/tty.h

这两个结构体实现了和tty关联的flip缓冲区。

tty\_driver

include/linux/tty\_driver.h

tty驱动和高层之间的编程接口。

tty\_ldisc

include/linux/tty\_ldisc.h

线路规程支持的入口函数。

表 6.4. 内核编程接口概述

内核接口

位置

描述

uart\_register\_driver()

drivers/serial/sderial\_core.c

向串行核心注册UART驱动。

uart\_add\_one\_port()

drivers/serial/sderial\_core.c

注册UART驱动支持的UART端口。

uart\_unregister\_driver()

drivers/serial/sderial\_core.c

从串行核心移除UART驱动。

platform\_device register()

platform\_device\_register\_simple()

platform\_add\_devices()

drivers/base/platform.c

注册platform设备。

platform\_device\_unregister()

drivers/base/platform.c

卸载platform设备。

platform\_driver\_register()/

platform\_driver\_unregister()

drivers/base/platform.c

注册/卸载platform驱动。

tty\_insert\_flip\_char()

include/linux/tty\_flip.h

向tty flip缓冲区添加一个字符。

tty\_flip\_buffer\_push()

drivers/char/tty\_io.c

排队一个将flip缓冲区推向线路规程的请求。

tty\_register\_driver()

drivers/char/tty\_io.c

向串行核心注册tty驱动。

tty\_unregister\_driver()

drivers/char/tty\_io.c

从串行核心注销tty驱动。

tty\_register\_ldisc()

drivers/char/tty\_io.c

通过注册指定的入口函数，创建线路规程。

tty\_unregister\_ldisc()

drivers/char/tty\_io.c

从串行核心移除线路规程。

一些串行数据的传送场景较为复杂。正如你在图6.3中所见，你可能需要混用和匹配多个不同的串行层模块。某些情况下，数据传送可能需要穿越多个线路规程来完成。例如，设置一个使用蓝牙建立的拨号连接就需要数据在HCI线路规程和PPP线路规程中传输。你可以尝试建立此连接，并使用内核调试器单步跟踪此代码流。

--------------------------------------------------------------------------------

[1] 平台设备通常不支持热插拔。对平台设备的probe()方法的调用不同于在后面章节中将学到的热插拔设备（如PCMCIA，PCI，和USB），它只是为了使驱动入口点的结构相同，以保持Linux设备模型一致、连续性。

--------------------------------------------------------------------------------

[MS1]此词可不译，是非常流行的说法，至今无准确的中文翻译，其含义基本只可以意会。