LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 14 章　Linux 终端设备驱动

在 Linux 系统中，终端设备非常重要，没有终端设备，系统将无法向用户反馈信息，Linux 系统中包含控制台、串口和伪终端 3 类终端设备。

14.1 节阐述了终端设备的概念及分类，14.2 节给出了 Linux 终端设备驱动的框架结构，重点讲解 tty_driver 结构体及其成员。

14.3～14.5 节在 14.2 节的基础上，分别讲解 Linux 终端设备驱动的模块加载/卸载函数和 open()、close()函数，数据读写流程及 tty 设备线路设置的编程方法。

在 Linux 系统中，串口驱动完全遵循 tty 驱动的框架结构，但是进行了底层操作的再次封装，14.6 节讲解了 Linux 系统针对串口 tty 驱动的这一封装，14.7 节则具体给出了串口 tty 驱动的实现方法。

14.8 节基于 14.6 节和 14.7 节的讲解给出了串口 tty 驱动的设计实例，即 S3C2410 集成 UART 的驱动。

14.1 终端设备

在 Linux 系统中，终端是一种字符型设备，它有多种类型，通常使用 tty 来简称各种类型的终端设备。tty 是 Teletype 的缩写，Teletype 是最早出现的一种终端设备，很像电传打字机，是由 Teletype 公司生产的。Linux 系统中包含如下几类终端设备。

1．串行端口终端（/dev/ttySn）

串行端口终端（Serial Port Terminal）是使用计算机串行端口连接的终端设备。计算机把每个串行端口都看作是一个字符设备。这些串行端口所对应的设备名称是/dev/ttyS0（或/dev/tts/0）、/dev/ttyS1（或/dev/tts/1）等，设备号分别是（4，0）、（4，1）等。

在命令行上把标准输出重定向到端口对应的设备文件名上就可以通过该端口发送数据，例如，在命令行提示符下输入“echo test > /dev/ttyS1”会把单词“test”发送到连接在 ttyS1 端口的设备上。

2．伪终端（/dev/pty/）

伪终端（Pseudo Terminal）是成对的逻辑终端设备，并存在成对的设备文件，如/dev/ptyp3 和/dev/ttyp3，它们与实际物理设备并不直接相关。如果一个程序把 ttyp3 看作是一个串行端口设备，则它对该端口的读/写操作会反映在该逻辑终端设备对应的 ttyp3 上，而 ttyp3 则是另一个程序用于读写操作的逻辑设备。这样，两个程序就可以通过这种逻辑设备进行通信，使用 ttyp3 的程序会认为自己正在与一个串行端口进行通信。

以 telnet 为例，如果某人在使用 telnet 程序连接到 Linux 系统，则 telnet 程序就可能会开始连接到设备 ptyp2 上，而此时一个 getty 程序会运行在对应的 ttyp2 端口上。当 telnet 从远端获取了一个字符时，该字符就会通过 ptyp2、ttyp2 传递给 getty 程序，而 getty 程序则会通过 ttyp2、ptyp2 和 telnet 程序返回“login:”字符串信息。这样，登录程序与 telnet 程序就通过伪终端进行通信。通过使用适当的软件，可以把两个或多个伪终端设备连接到同一个物理串行端口上。

3．控制台终端（/dev/ttyn，/dev/console）

如果当前进程有控制终端（Controlling Terminal），那么/dev/tty 就是当前进程的控制终端的设备特殊文件。可以使用命令“ps–ax”来查看进程与哪个控制终端相连，使用命令“tty”可以查看它具体对应哪个实际终端设备。/dev/tty 有些类似于到实际所使用终端设备的一个连接。

在 UNIX 系统中，计算机显示器通常被称为控制台终端（Console）。它仿真了类

型为 Linux 的一种终端（TERM = Linux），并且有一些设备特殊文件与之相关联：tty0、tty1、tty2 等。当用户在控制台上登录时，使用的是 tty1。按[Alt+F1]～[Alt+F6]组合键时，我们就可以切换到 tty2、tty3 等。tty1～tty6 等称为虚拟终端，而 tty0 则是当前所使用虚拟终端的一个别名，系统所产生的信息会发送到该终端上。因此不管当前正在使用哪个虚拟终端，系统信息都会发送到控制台终端上。用户可以登录到不同的虚拟终端上去，因而可以让系统同时有几个不同的会话期存在。只有系统或超级用户 root 可以向/dev/tty0 进行写操作。

在 Linux 系统中，可以在系统启动命令行里指定当前的输出终端，格式如下：

```
console=device, options
```

device 指代的是终端设备，可以是 tty0（前台的虚拟终端）、ttyX（第 X 个虚拟终端）、ttySX（第 X 个串口）、lp0（第一个并口）等。

options 指代对 device 进行的设置，它取决于具体的设备驱动。对于串口设备，参数用来定义为如下。

波特率、校验位、位数，格式为 BBBBPN，其中 BBBB 表示波特率，P 表示校验（n/o/e），N 表示位数，默认 options 是 9600n8。

用户可以在内核命令行中同时设定多个终端，这样输出将会在所有的终端上显示，而当用户调用 open()打开/dev/console 时，打开的将是设定的最后一个终端。例如：

```
console=ttyS1, 9600 console=tty0
```

定义了两个终端，而调用 open()打开/dev/console 时，将使用虚拟终端 tty0。但是内核消息会在 tty0 VGA 虚拟终端和串口 ttyS1 上同时显示。

通过查看/proc/tty/drivers 文件可以获知什么类型的 tty 设备存在以及什么驱动被加载到内核，这个文件包括一个当前存在的不同 tty 驱动的列表，包括驱动名、默认的节点名、驱动的主编号、这个驱动使用的次编号范围以及 tty 驱动的类型。例如，下面所示为一个/proc/tty/drivers 文件的例子。

```
[root@localhost root]# cat /proc/tty/drivers
/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
serial               /dev/ttyS       4 64-127 serial
pty_slave            /dev/pts      136 0-1048575 pty:slave
pty_master           /dev/ptm      128 0-1048575 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
```

14.2 终端设备驱动结构

Linux 内核中 tty 的层次结构如图 14.1 所示，包含 tty 核心、tty 线路规程和 tty 驱动，tty 线路规程的工作是以特殊的方式格式化从一个用户或者硬件收到的数据，这种格式化常常采用一个协议转换的形式，例如 PPP 和 Bluetooth。tty 设备发送数据的流程为：tty 核心从一个用户获取将要发送给一个 tty 设备的数据，tty 核心将数据传递给 tty 线路规程驱动，接着数据被传递到 tty 驱动，tty 驱动将数据转换为可以发送给硬件的格式。

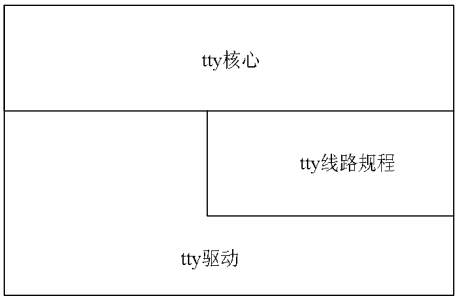

图 14.1　tty 分层结构

接收数据的流程为：从 tty 硬件接收到的数据向上交给 tty 驱动，进入 tty 线路规程驱动，再进入 tty 核心，在这里它被一个用户获取。尽管大多数时候 tty 核心和 tty 之间的数据传输会经历 tty 线路规程的转换，但是 tty 驱动与 tty 核心之间也可以直接传输数据。

图 14.2 显示了与 tty 相关的主要源文件及数据的流向。tty_io.c 定义了 tty 设备通用的 file_ operations 结构体并实现了接口函数 tty_register_driver()用于注册 tty 设备，它会利用 fs/char_dev.c 提供的接口函数注册字符设备，与具体设备对应的 tty 驱动将实现 tty_driver 结构体中的成员函数。同时 tty_io.c 也提供了 tty_register_ldisc()接口函数用于注册线路规程，n_tty.c 文件则实现了 tty_disc 结构体中的成员。

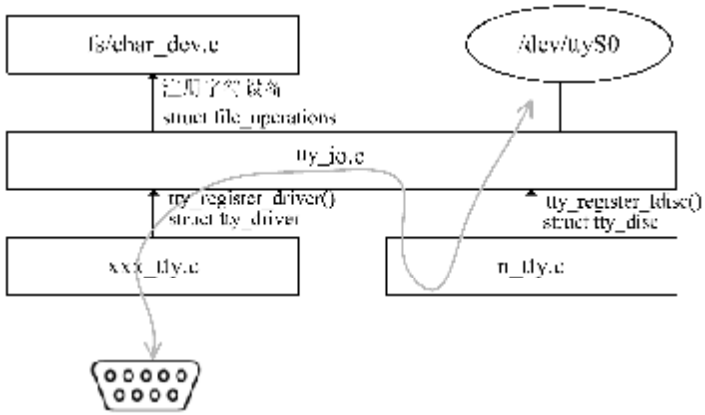

图 14.2　tty 主要源文件关系及数据流向

从图 14.2 可以看出，特定 tty 设备驱动的主体工作是填充 tty_driver 结构体中的成员，实现其中的成员函数，tty_driver 结构体的定义如代码清单 14.1 所示。

代码清单 14.1　tty_driver 结构体

```
1  struct tty_driver
2  {
3    int magic;
4    struct cdev cdev; /* 对应的字符设备 cdev */
5    struct module *owner;   /*这个驱动的模块拥有者 */
```

```
    6   const char *driver_name;
    7   const char *devfs_name;
    8   const char *name;   /* 设备名 */
    9   int name_base; /* offset of printed name */
    10  int major; /* 主设备号 */
    11  int minor_start; /* 开始次设备号 */
    12  int minor_num; /* 设备数量 */
    13  int num; /* 被分配的设备数量 */
    14  short type; /* tty 驱动的类型 */
    15  short subtype; /* tty 驱动的子类型 */
    16  struct termios init_termios; /* 初始线路设置 */
    17  int flags; /* tty 驱动标志 */
    18  int refcount; /*引用计数（针对可加载的 tty 驱动） */
    19  struct proc_dir_entry *proc_entry; /* /proc 文件系统入口 */
    20  struct tty_driver *other; /* 仅对 PTY 驱动有意义 */
    21  ...
    22  /* 接口函数 */
    23  int(*open)(struct tty_struct *tty, struct file *filp);
    24  void(*close)(struct tty_struct *tty, struct file *filp);
    25  int(*write)(struct tty_struct *tty, const unsigned char *buf, int
count);
    26  void(*put_char)(struct tty_struct *tty, unsigned char ch);
    27  void(*flush_chars)(struct tty_struct *tty);
    28  int(*write_room)(struct tty_struct *tty);
    29  int(*chars_in_buffer)(struct tty_struct *tty);
    30  int(*ioctl)(struct tty_struct *tty, struct file *file, unsigned int
cmd,
    31    unsigned long arg);
    32  void(*set_termios)(struct tty_struct *tty, struct termios *old);
    33  void(*throttle)(struct tty_struct *tty);
    34  void(*unthrottle)(struct tty_struct *tty);
    35  void(*stop)(struct tty_struct *tty);
    36  void(*start)(struct tty_struct *tty);
    37  void(*hangup)(struct tty_struct *tty);
    38  void(*break_ctl)(struct tty_struct *tty, int state);
    39  void(*flush_buffer)(struct tty_struct *tty);
    40  void(*set_ldisc)(struct tty_struct *tty);
    41  void(*wait_until_sent)(struct tty_struct *tty, int timeout);
    42  void(*send_xchar)(struct tty_struct *tty, char ch);
    43  int(*read_proc)(char *page, char **start, off_t off, int count, int
*eof,
    44    void *data);
    45   int(*write_proc)(struct file *file, const char __user *buffer,
unsigned long
    46    count, void *data);
    47  int(*tiocmget)(struct tty_struct *tty, struct file *file);
    48  int(*tiocmset)(struct tty_struct *tty, struct file *file, unsigned
int set,
    49    unsigned int clear);
    50
    51  struct list_head tty_drivers;
    52 };
```

tty_driver 结构体中的 magic 表示给这个结构体的“幻数”，设为 TTY_DRIVER_MAGIC，在 alloc_tty_driver()函数中被初始化。

name 与 driver_name 的不同在于后者表示驱动的名字，用在 /proc/tty 和 sysfs 中，而前者表示驱动的设备节点名。

type 与 subtype 描述 tty 驱动的类型和子类型，subtype 的值依赖于 type，type 成员的可能值为 TTY_DRIVER_TYPE_SYSTEM（由 tty 子系统内部使用，subtype 应当设为 SYSTEM_TYPE_TTY、SYSTEM_TYEP_CONSOLE、SYSTEM_TYPE_SYSCONS 或 SYSTEM_TYPE_SYSPTMX，这个类型不应当被任何常规 tty 驱动使用）、TTY_DRIVER_TYPE_CONSOLE（仅被控制台驱动使用）、TTY_DRIVER_TYPE_SERIAL（被任何串行类型驱动使用，subtype 应当设为 SERIAL_TYPE_ NORMAL 或 SERIAL_TYPE_CALLOUT）、TTY_DRIVER_TYPE_PTY（被伪控制台接口 pty 使用，此时 subtype 需要被设置为 PTY_TYPE_MASTER 或 PTY_TYPE_SLAVE）。

init_termios 为初始线路设置，为一个 termios 结构体，这个成员被用来提供一个线路设置集合。

termios 用于保存当前的线路设置，这些线路设置控制当前波特率、数据大小、数据流控设置等，这个结构体包含 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（线路规程类型）、cc_t c_cc[NCCS]（一个控制字符数组）等成员。

驱动会使用一个标准的数值集初始化这个成员，这个数值集来源于 tty_std_termios 变量，tty_std_termos 在 tty 核心中的定义如代码清单 14.2 所示。

代码清单 14.2　tty_std_termios 变量

```
1  struct termios tty_std_termios =
2  {
3   .c_iflag = ICRNL | IXON, /* 输入模式 */
4   .c_oflag = OPOST | ONLCR, /* 输出模式 */
5   .c_cflag = B38400 | CS8 | CREAD | HUPCL, /* 控制模式 */
6   .c_lflag = ISIG | ICANON | ECHO | ECHOE | ECHOK |
7   ECHOCTL | ECHOKE | IEXTEN,  /* 本地模式 */
8   .c_cc = INIT_C_CC  /* 控制字符，用来修改终端的特殊字符映射 */
9  };
```

tty_driver 结构体中的 major、minor_start、minor_num 表示主设备号、次设备号及可能的次设备数，name 表示设备名（如 ttyS），第 23～49 行的函数指针实际和 tty_operations 结构体等同，它们通常需在特定设备 tty 驱动模块初始化函数中被赋值。

put_char()为单字节写函数，当单个字节被写入设备时这个函数被 tty 核心调用，如果一个 tty 驱动没有定义这个函数，将使用 count 参数为 1 的 write()函数。

flush_chars()与 wait_until_sent()函数都用于刷新数据到硬件。

write_room()指示有多少缓冲区空闲，chars_in_buffer()指示缓冲区的长度。

当 ioctl(2)在设备节点上被调用时，ioctl()函数将被 tty 核心调用。

当设备的 termios 设置被改变时，set_termios()函数将被 tty 核心调用。

throttle()、unthrottle()、stop()和 start()用于控制 tty 核心的输入缓存。当 tty 核心的输入缓冲满时，throttle()函数将被调用，tty 驱动试图通知设备不应当发送字符给它。当 tty 核心的输入缓冲已被清空时，unthrottle()函数将被调用暗示设备可

以接收数据。stop()和 start()函数非常像 throttle()和 unthrottle()函数，但它们表示 tty 驱动应当停止发送数据给设备以及恢复发送数据。

当 tty 驱动挂起 tty 设备时，hangup()函数被调用，在此函数中进行相关的硬件操作。

当 tty 驱动要在 RS-232 端口上打开或关闭线路的 BREAK 状态时，break_ctl()线路中断控制函数被调用。如果 state 状态设为−1，BREAK 状态打开，如果状态设为 0，BREAK 状态关闭。如果这个函数由 tty 驱动实现，而 tty 核心将处理 TCSBRK、TCSBRKP、TIOCSBRK 和 TIOCCBRK 这些 ioctl 命令。

flush_buffer()函数用于刷新缓冲区并丢弃任何剩下的数据。

set_ldisc()函数用于设置线路规程，当 tty 核心改变 tty 驱动的线路规程时这个函数被调用，这个函数通常不需要被驱动定义。

send_xchar()为 X-类型字符发送函数，这个函数用来发送一个高优先级 XON 或者 XOFF 字符给 tty 设备，要被发送的字符在第二个参数 ch 中指定。

read_proc()和 write_proc()为/proc 文件系统的读和写函数。

tiocmget()函数用于获得 tty 设备的线路设置，对应的 tiocmset()用于设置 tty 设备的线路设置，参数 set 和 clear 包含了要设置或者清除的线路设置。

Linux 内核提供了一组函数用于操作 tty_driver 结构体及 tty 设备，如下所示。

1．分配 tty 驱动

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

这个函数返回 tty_driver 指针，其参数为要分配的设备数量，line 会被赋值给 tty_driver 的 num 成员，例如：

```
xxx_tty_driver = alloc_tty_driver(XXX_TTY_MINORS);
if (!xxx_tty_driver) //分配失败
 return -ENOMEM;
```

2．注册 tty 驱动

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

注册 tty 驱动成功时返回 0，参数为由 alloc_tty_driver ()分配的 tty_driver 结构体指针，例如：

```
retval = tty_register_driver(xxx_tty_driver);
if (retval) //注册失败
{
  printk(KERN_ERR "failed to register tiny tty driver");
  put_tty_driver(xxx_tty_driver);
  return retval;
}
```

3．注销 tty 驱动

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

这个函数与 tty_register_driver ()对应，tty 驱动最终会调用上述函数注销 tty_driver。

4．注册 tty 设备

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

仅有 tty_driver 是不够的，驱动必须依附于设备，tty_register_device()函数用于注册关联于 tty_driver 的设备，index 为设备的索引（范围是 0～driver->num），例如：

```
for (i = 0; i < XXX_TTY_MINORS; ++i)
tty_register_device(xxx_tty_driver, i, NULL);
```

5．注销 tty 设备

```
void tty_unregister_device(struct tty_driver *driver, unsigned index);
```

上述函数与 tty_register_device()对应，用于注销 tty 设备，其使用方法如下：

```
for (i = 0; i < XXX_TTY_MINORS; ++i)
tty_unregister_device(xxx_tty_driver, i);
```

6．设置 tty 驱动操作

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

上述函数会将 tty_operations 结构体中的函数指针复制到 tty_driver 对应的函数指针，在具体的 tty 驱动中，通常会定义一个设备特定的 tty_operations，tty_operations 的定义如代码清单 14.3 所示。tty_operations 中的成员函数与 tty_driver 中的同名成员函数意义完全一致，因此，这里不再赘述。

代码清单 14.3　tty_operations 结构体

```
1  struct tty_operations
2  {
3     int  (*open)(struct tty_struct * tty, struct file * filp);
4     void (*close)(struct tty_struct * tty, struct file * filp);
5     int  (*write)(struct tty_struct * tty,
6                 const unsigned char *buf, int count);
7     void (*put_char)(struct tty_struct *tty, unsigned char ch);
8     void (*flush_chars)(struct tty_struct *tty);
9     int  (*write_room)(struct tty_struct *tty);
10    int  (*chars_in_buffer)(struct tty_struct *tty);
11    int  (*ioctl)(struct tty_struct *tty, struct file * file,
12              unsigned int cmd, unsigned long arg);
13    void (*set_termios)(struct tty_struct *tty, struct termios * old);
14    void (*throttle)(struct tty_struct * tty);
15    void (*unthrottle)(struct tty_struct * tty);
```

```
16   void (*stop)(struct tty_struct *tty);
17   void (*start)(struct tty_struct *tty);
18   void (*hangup)(struct tty_struct *tty);
19   void (*break_ctl)(struct tty_struct *tty, int state);
20   void (*flush_buffer)(struct tty_struct *tty);
21   void (*set_ldisc)(struct tty_struct *tty);
22   void (*wait_until_sent)(struct tty_struct *tty, int timeout);
23   void (*send_xchar)(struct tty_struct *tty, char ch);
24   int (*read_proc)(char *page, char **start, off_t off,
25                  int count, int *eof, void *data);
26   int (*write_proc)(struct file *file, const char __user *buffer,
27                  unsigned long count, void *data);
28   int (*tiocmget)(struct tty_struct *tty, struct file *file);
29   int (*tiocmset)(struct tty_struct *tty, struct file *file,
30                    unsigned int set, unsigned int clear);
31 };
```

终端设备驱动都围绕 tty_driver 结构体而展开，一般而言，终端设备驱动应包含如下组成。

- 终端设备驱动模块加载函数和卸载函数：完成注册和注销 tty_driver，初始化和释放终端设备对应的 tty_driver 结构体成员及硬件资源。
- 实现 tty_operations 结构体中的一系列成员函数：主要是实现 open()、close()、write()、tiocmget()、tiocmset()等函数。

14.3 终端设备驱动的初始化与释放

14.3.1 模块加载与卸载函数

tty 驱动的模块加载函数中通常需要分配、初始化 tty_driver 结构体并申请必要的硬件资源，如代码清单 14.4 所示。tty 驱动的模块卸载函数完成与模块加载函数相反的工作。

代码清单 14.4　终端设备驱动的模块加载函数范例

```
1  /* tty 驱动的模块加载函数 */
2  static int __init xxx_init(void)
3  {
4    ...
5    /* 分配 tty_driver 结构体 */
6    xxx_tty_driver = alloc_tty_driver(XXX_PORTS);
7    /* 初始化 tty_driver 结构体 */
```

```
    8   xxx_tty_driver->owner = THIS_MODULE;
    9   xxx_tty_driver->devfs_name = "tts/";
   10   xxx_tty_driver->name = "ttyS";
   11   xxx_tty_driver->major = TTY_MAJOR;
   12   xxx_tty_driver->minor_start = 64;
   13   xxx_tty_driver->type = TTY_DRIVER_TYPE_SERIAL;
   14   xxx_tty_driver->subtype = SERIAL_TYPE_NORMAL;
   15   xxx_tty_driver->init_termios = tty_std_termios;
   16   xxx_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL
| CLOCAL;
   17   xxx_tty_driver->flags = TTY_DRIVER_REAL_RAW;
   18   tty_set_operations(xxx_tty_driver, &xxx_ops);
   19
   20   ret = tty_register_driver(xxx_tty_driver);
   21   if (ret)
   22   {
   23     printk(KERN_ERR "Couldn't register xxx serial driver\n");
   24     put_tty_driver(xxx_tty_driver);
   25     return ret;
   26   }
   27
   28   ...
   29   ret = request_irq(...); /* 硬件资源申请 */
   30   ...
   31 }
```

14.3.2　打开与关闭函数

当用户对 tty 驱动所分配的设备节点进行 open()系统调用时，tty_driver 中的 open()成员函数将被 tty 核心调用。tty 驱动必须设置 open()成员，否则，-ENODEV 将被返回给调用 open()的用户。

open()成员函数的第一个参数为一个指向分配给这个设备的 tty_struct 结构体的指针，第二个参数为文件指针。

tty_struct 结构体被 tty 核心用来保存当前 tty 端口的状态，它的大多数成员只被 tty 核心使用。tty_struct 中的几个重要成员如下。

- flags 标示 tty 设备的当前状态，包括 TTY_THROTTLED、TTY_IO_ERROR、TTY_OTHER_ CLOSED 、 TTY_EXCLUSIVE 、 TTY_DEBUG 、 TTY_DO_WRITE_WAKEUP、TTY_PUSH、TTY_CLOSING、TTY_DONT_FLIP、TTY_HW_COOK_OUT 、 TTY_HW_COOK_IN 、 TTY_ PTY_LOCK 、 TTY_NO_WRITE_SPLIT 等。
- ldisc 为给 tty 设备使用的线路规程。
- write_wait、read_wait 为 tty 写/读函数的等待队列，tty 驱动应当在合适的时机唤醒对应的等待队列。
- termios 为指向 tty 设备的当前 termios 设置的指针。
- stopped:1 指示是否停止 tty 设备，tty 驱动可以设置这个值；hw_stopped:1 指示是否 tty 设备已经被停止，tty 驱动可以设置这个值；flow_stopped:1 指示是否 tty 设备数据流停止。

- driver_data、disc_data 为数据指针，是用于存储 tty 驱动和线路规程的“私有”数据。

驱动中可以定义一个设备相关的结构体，并在 open()函数中将其赋值给 tty_struct 的 driver_data 成员，如代码清单 14.5 所示。

代码清单 14.5　在 tty 驱动打开函数中赋值 tty_struct 的 driver_data 成员

```
1  /* 设备“私有”数据结构体 */
2  struct xxx_tty
3  {
4    struct tty_struct *tty; /* tty_struct 指针 */
5    int open_count; /* 打开次数 */
6    struct semaphore sem; /* 结构体锁定信号量 */
7    int xmit_buf; /* 传输缓冲区 */
8    ...
9  }
10
11 /* 打开函数 */
12 static int xxx_open(struct tty_struct *tty, struct file *file)
13 {
14   struct xxx_tty *xxx;
15
16   /* 分配 xxx_tty */
17   xxx = kmalloc(sizeof(*xxx), GFP_KERNEL);
18   if (!xxx)
19     return  - ENOMEM;
20   /* 初始化 xxx_tty 中的成员 */
21   init_MUTEX(&xxx->sem);
22   xxx->open_count = 0;
23   ...
24   /* 使 tty_struct 中的 driver_data 指向 xxx_tty */
25   tty->driver_data = xxx;
26   xxx->tty = tty;
27   ...
28   return 0;
29 }
```

在用户对前面使用 open()系统调用而创建的文件句柄进行 close()系统调用时，tty_driver 中的 close()成员函数将被 tty 核心调用。

14.4 数据发送和接收

如图 14.3 所示为终端设备数据发送和接收过程中的数据流以及函数调用关系。用户在有数据发送给终端设备时，通过“write()系统调用—tty 核心—线路规程”的层层调用，最终调用 tty_driver 结构体中的 write()函数完成发送。

因为速度和 tty 硬件缓冲区容量的原因，不是所有的写程序要求的字符都可以在调用写函数时被发送，因此写函数应当返回能够发送给硬件的字节数以便用户程序检查是否所有的数据被真正写入。如果在 wirte()调用期间发生任何错误，一个负的错误码应当被返回。

tty_driver 的 write()函数接受 3 个参数：tty_struct、发送数据指针及要发送的字节

数，一般首先会通过 tty_struct 的 driver_data 成员得到设备私有信息结构体，然后依次进行必要的硬件操作开始发送，如代码清单 14.6 所示为 tty_driver 的 write()函数范例。

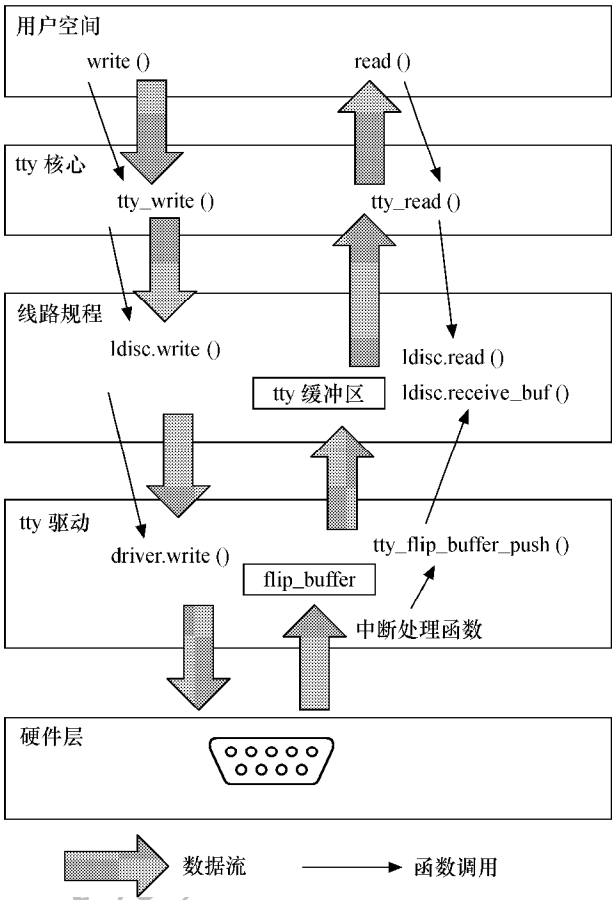

图 14.3　终端设备数据发送和接收过程中的数据流和函数调用关系

代码清单 14.6　tty_driver 结构体的 write()成员函数范例

```
    1  static int xxx_write(struct tty_struct *tty, const unsigned char *buf,
int count)
    2  {
    3    /* 获得 tty 设备私有数据 */
    4    struct xxx_tty *xxx = (struct xxx_tty*)tty->driver_data;
    5    ...
    6    /* 开始发送 */
    7    while (1)
    8    {
    9      local_irq_save(flags);
    10     c = min_t(int, count, min(SERIAL_XMIT_SIZE - xxx->xmit_cnt - 1,
    11       SERIAL_XMIT_SIZE - xxx->xmit_head));

    12     if (c <= 0)
    13     {
    14       local_irq_restore(flags);
    15       break;
    16     }
    17     //复制到发送缓冲区
```

```
    memcpy(xxx->xmit_buf + xxx->xmit_head, buf, c);
    xxx->xmit_head = (xxx->xmit_head + c) &(SERIAL_XMIT_SIZE - 1);
    xxx->xmit_cnt += c;
    local_irq_restore(flags);

    buf += c;
    count -= c;
    total += c;
  }

  if (xxx->xmit_cnt && !tty->stopped && !tty->hw_stopped)
  {
    start_xmit(xxx);//开始发送
  }
  return total; //返回发送的字节数
}
```

当 tty 子系统自己需要发送数据到 tty 设备时，如果没有实现 put_char()函数，write()函数将被调用，此时传入的 count 参数为 1，通过对代码清单 14.7 的分析即可获知。

代码清单 14.7　put_char()函数的 write()替代

```
int tty_register_driver(struct tty_driver *driver)
{
  ...
  if (!driver->put_char)//没有定义 put_char()函数
    driver->put_char = tty_default_put_char;
  ...
}
static void tty_default_put_char(struct tty_struct *tty, unsigned
char ch)
{
  tty->driver->write(tty, &ch, 1);//调用 tty_driver.write()函数
}
```

tty_driver 结构体中没有提供 read()函数。因为发送是用户主动的，而接收即用户调用 read()则是读一片缓冲区中已放好的数据。tty 核心在一个称为 struct tty_flip_buffer 的结构体中缓冲数据直到它被用户请求。因为 tty 核心提供了缓冲逻辑，因此每个 tty 驱动并非一定要实现它自身的缓冲逻辑。

tty 驱动不必过于关心 tty_flip_buffer 结构体的细节，如果其 count 字段大于或等于 TTY_ FLIP BUF_SIZE，这个 flip 缓冲区就需要被刷新到用户，刷新通过对 tty_flip_buffer_push()函数的调用来完成，代码清单 14.8 所示为 tty_flip_buffer_push()的使用范例。

代码清单 14.8　tty_flip_buffer_push()范例

```
for (i = 0; i < data_size; ++i)
{
  if (tty->flip.count >= TTY_FLIPBUF_SIZE)
    tty_flip_buffer_push(tty);//数据填满向上层“推”
```

```
5   tty_insert_flip_char(tty, data[i], TTY_NORMAL);//把数据插入缓冲区
6 }
7 tty_flip_buffer_push(tty);
```

从 tty 驱动接收到字符将被 tty_insert_flip_char()函数插入到 flip 缓冲区。该函数的第 1 个参数是数据应当保存入的 tty_struct 结构体，第 2 个参数是要保存的字符，第 3 个参数是应当为这个字符设置的标志，如果字符是一个接收到的常规字符，则设为 TTY_NORMAL，如果是一个特殊类型的指示错误的字符，依据具体的错误类型，应当设为 TTY_BREAK、TTY_PARITY 或 TTY_OVERRUN。

14.5 tty 线路设置

14.5.1　线路设置用户空间接口

用户可用如下两种方式改变 tty 设备的线路设置或者获取当前线路设置。

1．调用用户空间的 termios 库函数

用户空间的应用程序需引用 termios.h 头文件，该头文件包含了终端设备的 I/O 接口，实际是由 POSIX 定义的标准方法。对终端设备操作模式的描述由 termios 结构体完成，从代码清单 14.2 可以看出，这个结构体包含 c_iflag、c_oflag、c_cflag、c_lflag 和 c_cc[]几个成员。

termios 的 c_cflag 主要包含如下位域信息：CSIZE（字长）、CSTOPB（两个停止位）、 PARENB（奇偶校验位使能）、PARODD （奇校验位，当 PARENB 被使能时）、 CREAD（字符接收使能，如果没有置位，仍然从端口接收字符，但这些字符都要被丢弃）、 CRTSCTS （如果被置位，CTS 状态改变时将发送报告）、CLOCAL （如果没有置位，调制解调器状态改变时将发送报告）。

termios 的 c_iflag 主要包含如下位域信息：INPCK（使能帧和奇偶校验错误检查）、BRKINT（break 将清除终端输入/输出队列，向该终端上前台的程序发出 SIGINT 信号）、 PARMRK （奇偶校验和帧错误被标记，在 INPCK 被设置且 IGNPAR 未被设置的情况下才有意义）、IGNPAR （忽略奇偶校验和帧错误）、IGNBRK （忽略 break）。

通过 tcgetattr()、tcsetattr()函数即可完成对终端设备的操作模式的设置和获取，这两个函数的原型如下：

```
int tcgetattr (int fd, struct termios *termios_p);
int tcsetattr (int fd, int optional_actions, struct termios *termios_p);
```

例如，Raw 模式的线路设置如下。

- 非正规模式。
- 关闭回显。
- 禁止 CR 到 NL 的映射（ICRNL）、输入奇偶校验、输入第 8 位的截取（ISTRIP）以及输出流控制。
- 8 位字符（CS8），奇偶校验被禁止。
- 禁止所有的输出处理。

- 每次一个字节 （c_cc [VMIN] = 1，c_cc [VTIME] = 0）。

则对应的对 termios 结构体的设置就为：

```
termios_p->c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
                        | INLCR | IGNCR | ICRNL | IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
termios_p->c_cflag &= ~(CSIZE | PARENB);
termios_p->c_cflag |= CS8;
```

通过如下一组函数可完成输入/输出波特率的获取和设置：

```
speed_t cfgetospeed (struct termios *termios_p); //获得输出波特率
speed_t cfgetispeed (struct termios *termios_p); //获得输入波特率


int cfsetospeed (struct termios *termios_p, speed_t speed); //设置输出波特率
int cfsetispeed (struct termios *termios_p, speed_t speed); //设置输入波特率
```

如下一组函数则完成线路控制：

```
int tcdrain (int fd); //等待所有输出都被发送
int tcflush (int fd, int queue_selector); //flush 输入/输出缓存
int tcflow (int fd, int action); // 对输入和输出流进行控制
int tcsendbreak (int fd, int duration);//发送 break
```

tcflush 函数刷清（抛弃）输入缓存（终端驱动程序已接收到，但用户程序尚未读取）或输出缓存（用户程序已经写，但驱动尚未发送），queue 参数可取 TCIFLUSH（刷清输入队列）、TCOFLUSH（刷清输出队列）或 TCIOFLUSH（刷清输入、输出队列）。

tcflow()对输入输出进行流控制，action 参数可取 TCOOFF（输出被挂起）、TCOON（重新启动以前被挂起的输出）、TCIOFF（发送 1 个 STOP 字符，使终端设备暂停发送数据）、TCION（发送 1 个 START 字符，使终端恢复发送数据）。

tcsendbreak()函数在一个指定的时间区间内发送连续的二进位数 0。若 duration 参数为 0，则此种发送延续 0.25～0.5s。POSIX.1 说明若 duration 非 0，则发送时间依赖于实现。

2．对 tty 设备节点进行 ioctl()调用

大部分 termios 库函数会被转化为对 tty 设备节点的 ioctl()调用，例如 tcgetattr()、tcsetattr()函数对应着 TCGETS、TCSETS IO 控制命令。

TIOCMGET（获得 MODEM 状态位）、TIOCMSET（设置 MODEM 状态位）、TIOCMBIC （清除指示 MODEM 位）、TIOCMBIS（设置指示 MODEM 位）这 4 个 I/O 控制命令用于获取和设置 MODEM 握手，如 RTS、CTS、DTR、DSR、RI、CD 等。

14.5.2　tty 驱动的 set_termios 函数

大部分 termios 用户空间函数被库转换为对驱动节点的 ioctl()调用，而 tty ioctl 中的大部分命令会被 tty 核心转换为对 tty 驱动的 set_termios()函数的调用。set_termios()

函数需要根据用户对 termios 的设置（termios 设置包括字长、奇偶校验位、停止位、波特率等）完成实际的硬件设置。

tty_operations 中的 set_termios()函数原型为：

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

新的设置被保存在 tty_struct 中，旧的设置被保存在 old 参数中，若新旧参数相同，则什么都不需要做，对于被改变的设置，需完成硬件上的设置，代码清单 14.9 所示为 set_termios()函数的使用范例。

代码清单 14.9　tty 驱动程序 set_termios()函数范例

```
1  static void xxx_set_termios(struct tty_struct *tty, struct termios
*old_termios)
2  {
3    struct xxx_tty *info = (struct cyclades_port*)tty->driver_data;
4    /* 新设置等同于老设置，什么也不做 */
5    if (tty->termios->c_cflag == old_termios->c_cflag)
6      return ;
7    ...
8
9    /* 关闭 CRTSCTS 硬件流控制 */
10   if ((old_termios->c_cflag &CRTSCTS) && !(cflag &CRTSCTS))
11   {
12     ...
13   }
14
15   /* 打开 CRTSCTS 硬件流控制 */
16   if (!(old_termios->c_cflag &CRTSCTS) && (cflag &CRTSCTS))
17   {
18     ...
19   }
20
21   /* 设置字节大小 */
22   switch (tty->termios->c_cflag &CSIZE)
23   {
24
25     case CS5:
26     ...
27     case CS6:
28     ...
29     case CS7:
30     ...
31     case CS8:
```

```
32    ...
33  }
34
35  /* 设置奇偶校验 */
36  if (tty->termios->c_cflag &PARENB)
37    if (tty->termios->c_cflag &PARODD)  //奇校验
38    ...
39    else  //偶校验
40    ...
41  else //无校验
42  ...
43 }
```

14.5.3　tty 驱动的 tiocmget 和 tiocmset 函数

对 TIOCMGET、TIOCMSET、TIOCMBIC 和 TIOCMBIS IO 控制命令的调用将被 tty 核心转换为对 tty 驱动 tiocmget()函数和 tiocmset()函数的调用，TIOCMGET 对应 tiocmget()函数，TIOCMSET、TIOCMBIC 和 TIOCMBIS 对应 tiocmset()函数，分别用于读取 Modem 控制的设置和进行 Modem 的设置。代码清单 14.10 所示为 tiocmget()函数的范例，代码清单 14.11 所示为 tiocmset()函数的范例。

代码清单 14.10　tty 驱动程序的 tiocmget()函数范例

```
1  static int xxx_tiocmget(struct tty_struct *tty, struct file *file)
2  {
3    struct xxx_tty *info = tty->driver_ data;
4    unsigned int result = 0;
5    unsigned int msr = info->msr;
6    unsigned int mcr = info->mcr;
7    result = ((mcr &MCR_DTR) ? TIOCM_DTR : 0) |  /* DTR 被设置 */
8    ((mcr &MCR_RTS) ? TIOCM_RTS : 0) |  /* RTS 被设置 */
9    ((mcr &MCR_LOOP) ? TIOCM_LOOP : 0) |  /* LOOP 被设置 */
10   ((msr &MSR_CTS) ? TIOCM_CTS : 0) |  /* CTS 被设置 */
11   ((msr &MSR_CD) ? TIOCM_CAR : 0) |  /* CD 被设置*/
12   ((msr &MSR_RI) ? TIOCM_RI : 0) |  /* 振铃指示被设置 */
13   ((msr &MSR_DSR) ? TIOCM_DSR : 0); /* DSR 被设置 */
14   return result;
15 }
```

代码清单 14.11　tty 驱动程序的 tiocmset()函数范例

```
1  static int xxx_tiocmset(struct tty_struct *tty, struct file *file,
unsigned
2    int set, unsigned int clear)
```

```
3 {
4   struct xxx_tty *info = tty->driver_data;
5   unsigned int mcr = info->mcr;
6
7   if (set &TIOCM_RTS) /* 设置 RTS */
8     mcr |= MCR_RTS;
9   if (set &TIOCM_DTR) /* 设置 DTR */
10    mcr |= MCR_RTS;
11
12  if (clear &TIOCM_RTS) /* 清除 RTS */
13    mcr &= ~MCR_RTS;
14  if (clear &TIOCM_DTR) /* 清除 DTR */
15    mcr &= ~MCR_RTS;
16
17  /* 设置设备新的 MCR 值 */
18  tiny->mcr = mcr;
19  return 0;
20 }
```

tiocmget()函数会访问 MODEM 状态寄存器（MSR），而 tiocmset()函数会访问 MODEM 控制寄存器（MCR）。

14.5.4　tty 驱动的 ioctl 函数

当用户在 tty 设备节点上进行 ioctl(2)调用时，tty_operations 中的 ioctl()函数会被 tty 核心调用。如果 tty 驱动不知道如何处理传递给它的 ioctl 值，它返回–ENOIOCTLCMD，之后 tty 核心会执行一个通用的操作。

驱动中常见的需处理的 IO 控制命令包括 TIOCSERGETLSR（获得这个 tty 设备的线路状态寄存器 LSR 的值）、TIOCGSERIAL（获得串口线信息）、TIOCMIWAIT（等待 MSR 改变）、TIOCGICOUNT（获得中断计数）等。代码清单 14.12 所示为 tty 驱动程序的 ioctl()函数的范例。

代码清单 14.12　tty 驱动程序的 ioctl()函数范例

```
1  static int xxx_ioctl(struct tty_struct *tty, struct file *filp,
unsigned int
2   cmd, unsigned long arg)
3 {
4   struct xxx_tty *info = tty->driver_data;
5   ...
6   /* 处理各种命令 */
7   switch (cmd)
8   {
9     case TIOCGSERIAL:
10      ...
11    case TIOCSSERIAL:
```

```
12      ...
13    case TIOCSERCONFIG:
14      ...
15    case TIOCMIWAIT:
16    ...
17    case TIOCGICOUNT:
18    ...
19    case TIOCSERGETLSR:
20    ...
21  }
22  ...
23 }
```

14.6 UART 设备驱动

尽管一个特定的 UART 设备驱动完全可以遵循 14.2～14.5 节的方法来设计，即定义 tty_driver 并实现其中的成员函数，但是 Linux 内核已经在文件 serial_core.c 中实现了 UART 设备的通用 tty 驱动层（称为串口核心层），这样，UART 驱动的主要任务演变成实现 serial-core.c 中定义的一组 uart_xxx 接口而非 tty_xxx 接口，如图 14.4 所示。

serial_core.c 串口核心层完全可以被当作 14.2～14.5 节 tty 设备驱动的实例，它实现了 UART 设备的 tty 驱动。

Linux 驱动的这种分层思想在许多类型的设备驱动中都得到了体现，例如上一章 IDE 设备驱动中，内核实现了通用的 IDE 层用于处理块设备 I/O 请求，而具体的 IDE 则只需使用 ide_xxx 这样的接口，甚至不必理会复杂的块设备驱动结构。

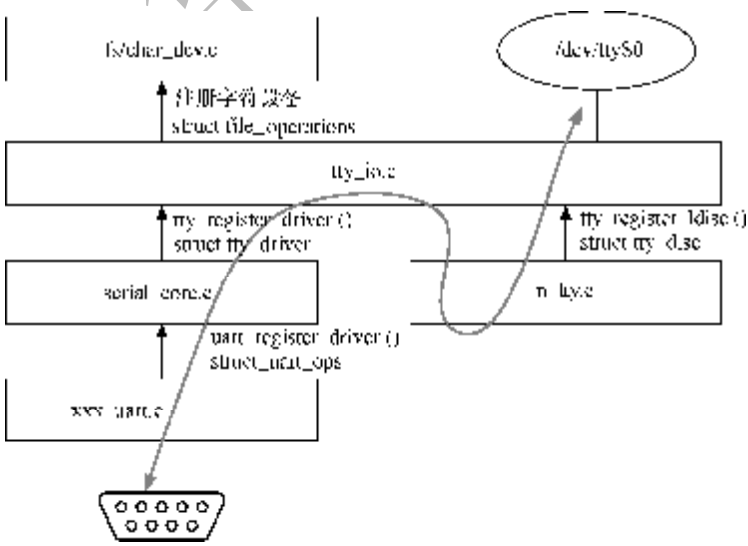

图 14.4　串口核心层

串口核心层为串口设备驱动提供了如下 3 个结构体。

1．uart_driver

uart_driver 包含串口设备的驱动名、设备名、设备号等信息，它封装了 tty_driver，使得底层的 UART 驱动无须关心 tty_driver，其定义如代码清单 14.13 所示。

代码清单 14.13　uart_driver 结构体

```
1  struct uart_driver
2  {
3    struct module *owner;
4    const char *driver_name;   //驱动名
5    const char *dev_name;      //设备名
6    const char *devfs_name;    //设备文件系统名
7    int major;                 //主设备号
8    int minor;                 //次设备号
9    int nr;
10   struct console *cons;
11
12   /* 私有的，底层驱动不应该访问这些成员，应该被初始化为 NULL */
13   struct uart_state *state;
14   struct tty_driver *tty_driver;
15 };
```

一个 tty 驱动必须注册/注销 tty_driver，而一个 UART 驱动则演变为注册/注销 uart_driver，使用如下接口：

```
int uart_register_driver(struct uart_driver *drv);
void uart_unregister_driver(struct uart_driver *drv);
```

实际上，uart_register_driver() 和 uart_unregister_driver() 中分别包含了 tty_register_driver()和 tty_unregister_driver()的操作，如代码清单 14.14 所示。

代码清单 14.14　uart_register_driver()和 uart_unregister_driver()函数

```
1  int uart_register_driver(struct uart_driver *drv)
2  {
3   struct tty_driver *normal = NULL;
4   int i, retval;
5   ...
6    /* 分配 tty_driver */
7   normal  = alloc_tty_driver(drv->nr);
8   if (!normal)
9       goto out;
10  drv->tty_driver = normal;
11   /* 初始化 tty_driver */
```

```
    normal->owner           = drv->owner;
    normal->driver_name = drv->driver_name;
    normal->devfs_name  = drv->devfs_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->flags           =          TTY_DRIVER_REAL_RAW          |
TTY_DRIVER_NO_DEVFS;
    normal->driver_state    = drv;
    tty_set_operations(normal, &uart_ops);

    ...
    /* 注册 tty 驱动 */
    retval = tty_register_driver(normal);
    out:
    if (retval < 0) {
        put_tty_driver(normal);
        kfree(drv->state);
    }
    return retval;
}

void uart_unregister_driver(struct uart_driver *drv)
{
    struct tty_driver *p = drv->tty_driver;
    tty_unregister_driver(p);  /* 注销 tty 驱动 */
    put_tty_driver(p);
    kfree(drv->state);
    drv->tty_driver = NULL;
}
```

2．uart_port

uart_port 用于描述一个 UART 端口（直接对应于一个串口）的 I/O 端口或 I/O 内存地址、FIFO 大小、端口类型等信息，其定义如代码清单 14.15 所示。

代码清单 14.15　uart_port 结构体

```
1  struct uart_port
2  {
3    spinlock_t lock; /* 端口锁 */
4    unsigned int iobase; /* I/O 端口基地址 */
5    unsigned char __iomem *membase; /* I/O 内存基地址 */
6    unsigned int irq; /* 终端号 */
7    unsigned int uartclk; /* UART 时钟 */
8    unsigned char fifosize; /* 传输 fifo 大小 */
9    unsigned char x_char; /* xon/xoff 字符 */
10   unsigned char regshift; /* 寄存器位移 */
11   unsigned char iotype; /* I/O 存取类型 */
12
13   #define UPIO_PORT      (0)  /* I/O 端口*/
14   #define UPIO_HUB6      (1)
15   #define UPIO_MEM       (2)  /* I/O 内存*/
16   #define UPIO_MEM32     (3)
17   #define UPIO_AU            (4)         /* Au1x00 类型 IO */
18
19   unsigned int read_status_mask; /* 驱动特定的读状态掩码 */
20   unsigned int ignore_status_mask; /* 驱动特定的忽略状态掩码 */
21   struct uart_info *info; /* 指向 parent 信息 */
22   struct uart_icount icount; /* 计数 */
23
24   struct console *cons; /* console 结构体 */
25   #ifdef CONFIG_SERIAL_CORE_CONSOLE
26     unsigned long sysrq; /* sysrq 超时 */
27   #endif
28
29   upf_t flags;
30
31   #define UPF_FOURPORT          ((__force upf_t) (1 << 1))
32   #define UPF_SAK               ((__force upf_t) (1 << 2))
33   #define UPF_SPD_MASK          ((__force upf_t) (0x1030))
34   #define UPF_SPD_HI        ((__force upf_t) (0x0010))
35   #define UPF_SPD_VHI           ((__force upf_t) (0x0020))
36   #define UPF_SPD_CUST          ((__force upf_t) (0x0030))
37   #define UPF_SPD_SHI           ((__force upf_t) (0x1000))
38   #define UPF_SPD_WARP          ((__force upf_t) (0x1010))
39   #define UPF_SKIP_TEST         ((__force upf_t) (1 << 6))
40   #define UPF_AUTO_IRQ          ((__force upf_t) (1 << 7))
41   #define UPF_HARDPPS_CD    ((__force upf_t) (1 << 11))
42   #define UPF_LOW_LATENCY       ((__force upf_t) (1 << 13))
43   #define UPF_BUGGY_UART    ((__force upf_t) (1 << 14))
44   #define UPF_MAGIC_MULTIPLIER  ((__force upf_t) (1 << 16))
45   #define UPF_CONS_FLOW         ((__force upf_t) (1 << 23))
46   #define UPF_SHARE_IRQ         ((__force upf_t) (1 << 24))
47   #define UPF_BOOT_AUTOCONF     ((__force upf_t) (1 << 28))
48   #define UPF_IOREMAP           ((__force upf_t) (1 << 31))
```

```
49
50  #define UPF_CHANGE_MASK       ((__force upf_t) (0x17fff))
51  #define UPF_USR_MASK          ((__force upf_t)
52     (UPF_SPD_MASK|UPF_LOW_LATENCY))
53  unsigned int mctrl; /* 目前 modem 控制设置 */
54  unsigned int timeout; /* 基于字符的超时 */
55  unsigned int type; /* 端口类型 */
56  const struct uart_ops *ops; /* UART 操作集 */
57  unsigned int custom_divisor;
58  unsigned int line; /* 端口索引 */
59  unsigned long mapbase; /* ioremap 后基地址 */
60  struct device *dev; /* parent 设备 */
61  unsigned char hub6;
62  unsigned char unused[3];
63 };
```

串口核心层提供如下函数来添加一个端口：

```
int uart_add_one_port(struct uart_driver *drv, struct uart_port *port);
```

对上述函数的调用应该发生在 uart_register_driver()之后，uart_add_one_port()的一个最重要作用是封装了 tty_register_device()。

uart_add_one_port() 的“反函数”是 uart_remove_one_port()，其中会调用 tty_unregister_device()，原型为：

```
int uart_remove_one_port(struct uart_driver *drv, struct uart_port
*port);
```

驱动中虽然不需要处理 uart_port 的 uart_info 成员，但是在发送时，从用户来的数据被保存在 xmit（被定义为 circ_buf，即环形缓冲区）中，因此 UART 驱动在发送数据时（一般在发送中断处理函数中），需要从这个 circ_buf 获取上层传递下来的字符。

3．uart_ops

uart_ops 定义了对 UART 的一系列操作，包括发送、接收及线路设置等，如果说 tty_driver 中的 tty_operations 对于串口还较为抽象，那么 uart_ops 则直接面向了串口的 UART，其定义如代码清单 14.16 所示。Linux 驱动的这种层次非常类似于面向对象编程中基类、派生类的关系，派生类针对特定的事物会更加具体，而基类则处于更高的抽象层次上。

代码清单 14.16　uart_ops 结构体

```
1  struct uart_ops
2  {
3    unsigned int(*tx_empty)(struct uart_port*);
4    void(*set_mctrl)(struct uart_port *, unsigned int mctrl);
5    unsigned int(*get_mctrl)(struct uart_port*);
6    void(*stop_tx)(struct uart_port*);            //停止发送
7    void(*start_tx)(struct uart_port*);           //开始发送
```

```
    8    void(*send_xchar)(struct uart_port *, char ch); //发送 xchar
    9    void(*stop_rx)(struct uart_port*);            //停止接收
    10   void(*enable_ms)(struct uart_port*);
    11   void(*break_ctl)(struct uart_port *, int ctl);
    12   int(*startup)(struct uart_port*);
    13   void(*shutdown)(struct uart_port*);
    14   void(*set_termios)(struct uart_port *, struct termios *new, struct
termios
    15     *old);       //设置 termios
    16    void(*pm)(struct uart_port *, unsigned int state, unsigned int
oldstate);
    17   int(*set_wake)(struct uart_port *, unsigned int state);
    18
    19   /* 返回一个描述端口类型的字符串 */
    20   const char *(*type)(struct uart_port*);
    21
    22   /* 释放端口使用的 I/O 和内存资源，必要的情况下，应该进行 iounmap 操作 */
    23   void(*release_port)(struct uart_port*);
    24   /* 申请端口使用的 I/O 和内存资源 */
    25   int(*request_port)(struct uart_port*);
    26
    27   void(*config_port)(struct uart_port *, int);
    28   int(*verify_port)(struct uart_port *, struct serial_struct*);
    29   int(*ioctl)(struct uart_port *, unsigned int, unsigned long);
    30 };
```

serial_core.c 中定义了 tty_operations 的实例，包含 uart_open()、uart_close()、uart_write()、uart_send_xchar()等成员函数（如代码清单 14.17 所示），这些函数会借助 uart_ops 结构体中的成员函数来完成具体的操作，代码清单 14.18 所示 tty_operations 的 uart_send_xchar()成员函数利用 uart_ops 中 start_tx()、send_xchar()成员函数的例子。

代码清单 14.17　串口核心层的 tty_operations 实例

```
    1  static struct tty_operations uart_ops =
    2  {
    3   .open       = uart_open,//串口打开
    4   .close      = uart_close,//串口关闭
    5   .write      = uart_write,//串口发送
    6   .put_char   = uart_put_char,//...
    7   .flush_chars    = uart_flush_chars,
    8   .write_room = uart_write_room,
    9   .chars_in_buffer= uart_chars_in_buffer,
    10  .flush_buffer   = uart_flush_buffer,
    11  .ioctl      = uart_ioctl,
    12  .throttle   = uart_throttle,
    13  .unthrottle = uart_unthrottle,
    14  .send_xchar = uart_send_xchar,
    15  .set_termios    = uart_set_termios,
    16  .stop       = uart_stop,
    17  .start      = uart_start,
    18  .hangup     = uart_hangup,
    19  .break_ctl  = uart_break_ctl,
    20  .wait_until_sent= uart_wait_until_sent,
    21 #ifdef CONFIG_PROC_FS
```

```
22  .read_proc  = uart_read_proc, //proc 入口读函数
23 #endif
24  .tiocmget   = uart_tiocmget,
25  .tiocmset   = uart_tiocmset,
26 };
```

代码清单 14.18　串口核心层的 tty_operations 与 uart_ops 关系

```
1  static void uart_send_xchar(struct tty_struct *tty, char ch)
2  {
3    struct uart_state *state = tty->driver_data;
4    struct uart_port *port = state->port;
5    unsigned long flags;
6    //如果 uart_ops 中实现了 send_xchar 成员函数
7    if (port->ops->send_xchar)
8      port->ops->send_xchar(port, ch);
9    else //uart_ops 中未实现 send_xchar 成员函数
10   {
11     port->x_char = ch; //xchar 赋值
12     if (ch)
13     {
14       spin_lock_irqsave(&port->lock, flags);
15       port->ops->start_tx(port);  //发送 xchar
16       spin_unlock_irqrestore(&port->lock, flags);
17     }
18   }
19 }
```

在使用串口核心层这个通用串口 tty 驱动层的接口后，一个串口驱动要完成的主要工作如下。

- 定义 uart_driver、uart_ops、uart_port 等结构体的实例并在适当的地方根据具体硬件和驱动的情况初始化它们，当然具体设备 xxx 的驱动可以将这些结构套在新定义的 xxx_uart_driver、xxx_uart_ops、xxx_uart_port 之内。
- 在模块初始化时调用 uart_register_driver()和 uart_add_one_port()以注册 UART 驱动并添加端口，在模块卸载时调用 uart_unregister_driver()和 uart_remove_one_port()以注销 UART 驱动并移除端口。
- 根据具体硬件的 datasheet 实现 uart_ops 中的成员函数，这些函数的实现成为 UART 驱动的主体工作。

14.7 S3C2410 UART 的驱动实例

14.7.1　S3C2410 串口硬件描述

S3C2410 内部具有 3 个独立的 UART 控制器，每个控制器都可以工作在 Interrupt（中断）模式或 DMA（直接内存访问）模式，也就是说 UART 控制器可以在 CPU 与 UART 控制器传送资料的时候产生中断或 DMA 请求。S3C2410 集成的每个 UART 均具有 16 字节的 FIFO，支持的最高波特率可达到 230.4Kbit/s。

ULCONn（UART Line Control Register）寄存器用于 S3C2410 UART 的线路控制，

用于设置模式、每帧的数据位数、停止位数及奇偶校验，如表 14.1 所示。

表 14.1　　S3C2410 UART 的 ULCONn 寄存器

ULCONn	位	描　　述
保留	[7]	
红外模式	[6]	0：正常模式 1：红外模式
奇偶校验	[5:3]	0xx：无校验，100：奇校验，101：偶校验
停止位	[2]	0：1 个停止位，1：2 个停止位
字长	[1:0]	00：5 位，01：6 位，10：7 位，11：8 位

UCONn（UART Control Register）寄存器用于从整体上控制 S3C2410 UART 的中断模式及工作模式（DMA、中断、轮询）等，如表 14.2 所示。

表 14.2　　S3C2410 UART 的 UCONn 寄存器

UCONn	位	描　　述
时钟选择	[10]	为 UART 的波特率产生选择 PCLK 或 UCLK 时钟
Tx 中断	[9]	0：脉冲，1：电平
Rx 中断	[8]	0：脉冲，1：电平
Rx 超时使能	[7]	当 UART 被使能，使能/禁止 Rx 超时中断 0：禁止，1：使能
Rx 错误状态中断使能	[6]	使能接收异常中断（如 break、帧错误、校验错、溢出等）
loopback	[5]	0：正常模式，1：回环
发送 break	[4]	设置该位将造成 UART 在 1 帧的时间内发送 break，当发送完 break 后，该位将自动被清除
发送模式	[3:2]	发送数据到 UART 的模式，00：禁止，01：中断或轮询， 10：DMA0（仅针对 UART0）、DMA3（仅针对 UART3）， 11：DMA1（仅针对 UART1）
接收模式	[1:0]	从 UART 接收数据的模式，00：禁止，01：中断或轮询， 10：DMA0（仅针对 UART0）

UFCONn （UART FIFO Conrtol Register）寄存器用于 S3C2410 UART 的 FIFO 控制，用于控制 FIFO 中断的触发级别以及复位时是否清空 FIFO 中的内容，如表 14.3 所示。

表 14.3　　S3C2410 UART 的 UFCONn 寄存器

UFCONn	位	描　　述
Tx FIFO 触发级别	[7:6]	决定发送 FIFO 的触发级别 00：空，01：4 字节，10：8 字节，11：12 字节
Rx FIFO 触发级别	[5:4]	决定接收 FIFO 的触发级别 00：4 字节，01：8 字节，10：12 字节，11：16 字节
Tx FIFO 复位	[2]	复位 FIFO 后自动清除 FIFO 0：正常，1：Tx FIFO 复位
Rx FIFO 复位	[1]	复位 FIFO 后自动清除 FIFO 0：正常，1：Tx FIFO 复位
FIFO 使能	[0]	0：禁止，1：使能

代码清单 14.19 所示为 UFCONn 寄存器的位掩码和默认设置（使能 FIFO、Tx FIFO 为空时触发中断，Rx FIFO 中包含 8 个字节时触发中断）。

代码清单 14.19　S3C2410 UART UFCONn 寄存器的位掩码和默认设置

```
1  #define S3C2410_UFCON_FIFOMODE    (1<<0)
2  #define S3C2410_UFCON_TXTRIG0     (0<<6)
3  #define S3C2410_UFCON_RXTRIG8     (1<<4)
4  #define S3C2410_UFCON_RXTRIG12    (2<<4)
5
6  #define S3C2410_UFCON_RESETBOTH   (3<<1)
7  #define S3C2410_UFCON_RESETTX     (1<<2)
8  #define S3C2410_UFCON_RESETRX     (1<<1)
9
10 #define S3C2410_UFCON_DEFAULT     (S3C2410_UFCON_FIFOMODE | \
11                 S3C2410_UFCON_TXTRIG0  | \
12                 S3C2410_UFCON_RXTRIG8 )
```

UFSTATn（UART FIFO Status Register）寄存器用于表示 UART FIFO 的状态，如表 14.4 所示。

表 14.4　　S3C2410 UART 的 UFSTATn 寄存器

UFSTATn	位	描　　述
保留	[15:10]	
Tx FIFO 满	[9]	当 Tx FIFO 满后，将自动被设置为 1 0：0 字节 ≤ Tx FIFO 中数据的数量≤ 15 1：Tx FIFO 中数据的数量= 15
Rx FIFO 满	[8]	当 Rx FIFO 满后，将自动被设置为 1 0：0 字节 ≤ Rx FIFO 中数据的数量≤ 15 1：Tx FIFO 中数据的数量 = 15
Tx FIFO 中数据的数量	[7:4]	

Rx FIFO 中数据的数量	[3:0]	

由于 UFSTATn 寄存器中的 Tx FIFO 中数据的数量和 Rx FIFO 中数据的数量分别占据 [7:4] 和 [3:0] 位，因此定义 S3C2410_UFSTAT_TXSHIFT 和 S3C2410_UFSTAT_RXSHIFT 分别为 4 和 0，代码清单 14.20 所示为 UFSTATn 寄存器的位掩码等信息。

代码清单 14.20　S3C2410 UART UFSTATn 寄存器的位掩码

```
1 #define S3C2410_UFSTAT_TXFULL  (1<<9)
2 #define S3C2410_UFSTAT_RXFULL  (1<<8)
3 #define S3C2410_UFSTAT_TXMASK  (15<<4)
4 #define S3C2410_UFSTAT_TXSHIFT     (4)
5 #define S3C2410_UFSTAT_RXMASK  (15<<0)
6 #define S3C2410_UFSTAT_RXSHIFT     (0)
```

UTXHn（UART Transmit Buffer Register）和 URXHn（UART Receive Buffer Register）分别是 UART 发送和接收数据寄存器，这两个寄存器存放着发送和接收的数据。

UTRSTATn（UART TX/RX Status Register）寄存器反映了发送和接收的状态，通过这个寄存器，驱动程序可以判断 URXHn 中是否有数据接收到或 UTXHn 是否为空，这个寄存器主要在非 FIFO 模式时使用。

UMCONn（UART Modem Control Register）用于 S3C2410 UART 的 modem 控制，设置是否使用 RTS 流控，若采用流控，可选择自动流控（Auto Flow Control，AFC）或由软件控制 RTS 信号的“高”或“低”电平。

14.7.2　S3C2410 串口驱动的数据结构

S3C2410 串口驱动中 uart_driver 结构体实例的定义如代码清单 14.21 所示，设备名为“s3c2410_serial”，驱动名为“ttySAC”。

代码清单 14.21　S3C2410 串口驱动的 uart_driver 结构体

```
1  #define S3C24XX_SERIAL_NAME  "ttySAC"
2  #define S3C24XX_SERIAL_DEVFS    "tts/"
3  #define S3C24XX_SERIAL_MAJOR 204
4  #define S3C24XX_SERIAL_MINOR 64
5
6  static struct uart_driver s3c24xx_uart_drv =
7  {
8    .owner      = THIS_MODULE,
9    .dev_name   = "s3c2410_serial",
10   .nr     = 3,
11   .cons       = S3C24XX_SERIAL_CONSOLE,
12   .driver_name    = S3C24XX_SERIAL_NAME,
13   .devfs_name = S3C24XX_SERIAL_DEVFS,
14   .major      = S3C24XX_SERIAL_MAJOR,
```

```
15  .minor      = S3C24XX_SERIAL_MINOR,
16 };
```

S3C2410 串口驱动中定义了结构体 s3c24xx_uart_port，该结构体中封装了 uart_port 结构体及一些针对 S3C2410 UART 的附加信息，代码清单 14.22 所示为 s3c24xx_uart_port 结构体及其实例 s3c24xx_serial_ports[]数组。

代码清单 14.22　S3C2410 串口驱动的 s3c24xx_uart_port 结构体

```
1  struct s3c24xx_uart_port
2  {
3   unsigned char          rx_claimed;
4   unsigned char          tx_claimed;
5
6   struct s3c24xx_uart_info   *info;
7   struct s3c24xx_uart_clksrc *clksrc;
8   struct clk         *clk;
9   struct clk         *baudclk;
10  struct uart_port       port;
11 };
12
13 static struct s3c24xx_uart_port s3c24xx_serial_ports[NR_PORTS] = {
14  [0] = {
15      .port = {
16          .lock       = SPIN_LOCK_UNLOCKED,
17          .iotype     = UPIO_MEM,
18          .irq            = IRQ_S3CUART_RX0,
19          .uartclk        = 0,
20          .fifosize   = 16,
21          .ops            = &s3c24xx_serial_ops,
22          .flags      = UPF_BOOT_AUTOCONF,
23          .line       = 0,//端口索引: 0
24      }
25  },
26  [1] = {
27      .port = {
28          .lock       = SPIN_LOCK_UNLOCKED,
29          .iotype     = UPIO_MEM,
30          .irq            = IRQ_S3CUART_RX1,
31          .uartclk        = 0,
32          .fifosize   = 16,
33          .ops            = &s3c24xx_serial_ops,
```

```
34          .flags      = UPF_BOOT_AUTOCONF,
35          .line       = 1, //端口索引: 1
36      }
37  },
38 #if NR_PORTS > 2
39
40  [2] = {
41      .port = {
42          .lock       = SPIN_LOCK_UNLOCKED,
43          .iotype     = UPIO_MEM,
44          .irq            = IRQ_S3CUART_RX2,
45          .uartclk        = 0,
46          .fifosize   = 16,
47          .ops            = &s3c24xx_serial_ops,
48          .flags      = UPF_BOOT_AUTOCONF,
49          .line       = 2, //端口索引: 2
50      }
51  }
52 #endif
53 };
```

S3C2410 串口驱动中 uart_ops 结构体实例的定义如代码清单 14.23 所示，将一系列 s3c24xx_ serial_函数赋值给 uart_ops 结构体的成员。

代码清单 14.23　S3C2410 串口驱动的 uart_ops 结构体

```
1  static struct uart_ops s3c24xx_serial_ops =
2  {
3    .pm     = s3c24xx_serial_pm,
4    .tx_empty   = s3c24xx_serial_tx_empty,//发送缓冲区空
5    .get_mctrl  = s3c24xx_serial_get_mctrl,//得到 modem 控制设置
6    .set_mctrl  = s3c24xx_serial_set_mctrl, //设置 modem 控制（MCR）
7    .stop_tx    = s3c24xx_serial_stop_tx, //停止接收字符
8    .start_tx   = s3c24xx_serial_start_tx,//开始传输字符
9    .stop_rx    = s3c24xx_serial_stop_rx, //停止接收字符
10   .enable_ms  = s3c24xx_serial_enable_ms,// modem 状态中断使能
11   .break_ctl  = s3c24xx_serial_break_ctl,// 控制 break 信号的传输
12   .startup    = s3c24xx_serial_startup,//启动端口
13   .shutdown   = s3c24xx_serial_shutdown,// 禁用端口
14   .set_termios    = s3c24xx_serial_set_termios,//改变端口参数
15   .type       = s3c24xx_serial_type,//返回描述特定端口的常量字符串指针
16   .release_port   = s3c24xx_serial_release_port,//释放端口占用的内存和
I/O 资源
```

```
    17  .request_port   = s3c24xx_serial_request_port,//申请端口所需的内存和
I/O 资源
    18  .config_port    = s3c24xx_serial_config_port,//执行端口所需的自动配
置步骤
    19  .verify_port    = s3c24xx_serial_verify_port,//验证新的串行端口信息
    20 };
```

set_mctrl()函数的原型为：

```
    void (*set_mctrl)(struct uart_port *port, u_int mctrl);
```

它将参数 port 所对应的调制解调器控制线的值设为参数 mctrl 的值。

get_mctrl()函数的原型为：

```
    unsigned int (*get_mctrl)(struct uart_port *port);
```

该函数返回调制解调器控制输入的现有状态，这些状态信息包括：TIOCM_CD（CD 信号状态）、TIOCM_CTS（CTS 信号状态）、TIOCM_DSR（DSR 信号状态）、TIOCM_RI（RI 信号状态）等。如果信号被置为有效，则对应位将被置位。

端口启动函数 startup()的原型为：

```
    int (*startup)(struct uart_port *port, struct uart_info *info);
```

该函数申请所有中断资源，初始化底层驱动状态，并开启端口为可接收数据的状态。

shutdown()函数实现与 startup()函数相反的作用，其原型为：

```
    void (*shutdown)(struct uart_port *port, struct uart_info *info);
```

这个函数禁用端口，释放所有的中断资源。

回过头来看 s3c24xx_uart_port 结构体，其中的 s3c24xx_uart_info 成员（代码清单 14.22 第 6 行）是一些针对 S3C2410 UART 的信息，其定义如代码清单 14.24 所示。

代码清单 14.24　S3C2410 串口驱动的 s3c24xx_uart_info 结构体

```
1  static struct s3c24xx_uart_info s3c2410_uart_inf =
2  {
3   .name       = "Samsung S3C2410 UART",
4   .type       = PORT_S3C2410,
5   .fifosize   = 16,
6   .rx_fifomask    = S3C2410_UFSTAT_RXMASK,
7   .rx_fifoshift   = S3C2410_UFSTAT_RXSHIFT,
8   .rx_fifofull    = S3C2410_UFSTAT_RXFULL,
9   .tx_fifofull    = S3C2410_UFSTAT_TXFULL,
10  .tx_fifomask    = S3C2410_UFSTAT_TXMASK,
11  .tx_fifoshift   = S3C2410_UFSTAT_TXSHIFT,
12  .get_clksrc = s3c2410_serial_getsource,
13  .set_clksrc = s3c2410_serial_setsource,
14  .reset_port = s3c2410_serial_resetport,
15 };
```

在 S3C2410 串口驱动中，针对 UART 的设置（UCONn、ULCONn、UFCONn 寄存器等）被封装到 s3c2410_uartcfg 结构体中，其定义如代码清单 14.25 所示。

代码清单 14.25　S3C2410 串口驱动 s3c2410_uartcfg 结构体

```
1  struct s3c2410_uartcfg
2  {
3    unsigned char hwport; /* 硬件端口号 */
4    unsigned char unused;
5    unsigned short flags;
6    unsigned long uart_flags; /* 默认的 UART 标志 */
7
8    unsigned long ucon; /* 端口的 ucon 值 */
9    unsigned long ulcon; /* 端口的 ulcon 值 */
10   unsigned long ufcon; /* 端口的 ufcon 值 */
11
12   struct s3c24xx_uart_clksrc *clocks;
13   unsigned int clocks_size;
14 };
```

14.7.3　S3C2410 串口驱动的初始化与释放

在 S3C2410 串口驱动的模块加载函数中会调用 uart_register_driver()注册 s3c24xx_uart_drv 这个 uart_driver，同时经过 s3c2410_serial_init()→s3c24xx_serial_init()→ platform_driver_register() 的调用导致 s3c24xx_serial_probe() 被执行，而 s3c24xx_serial_probe()函数中会调用 s3c24xx_serial_ init_port()初始化 UART 端口，并调用 uart_add_one_port()添加端口，整个过程的对应代码如清单 14.26 所示。

代码清单 14.26　S3C2410 串口驱动的初始化过程

```
1   static int __init s3c24xx_serial_modinit(void)
2   {
3   int ret;
4     //注册 uart_driver
5   ret = uart_register_driver(&s3c24xx_uart_drv);
6   if (ret < 0) {
7       printk(KERN_ERR "failed to register UART driver\n");
8       return -1;
9   }
10    //初始化 s3c2410 的串口
11  s3c2410_serial_init();
12
13  return 0;
14  }
15
16  static inline int s3c2410_serial_init(void)
17  {
18  return                  s3c24xx_serial_init(&s3c2410_serial_drv,
&s3c2410_uart_inf);
19  }
20
```

```
21  static int s3c24xx_serial_init(struct platform_driver *drv,
22                struct s3c24xx_uart_info *info)
23  {
24  dbg("s3c24xx_serial_init(%p,%p)\n", drv, info);
25  return platform_driver_register(drv);//注册平台驱动
26  }
27
28  //平台驱动的 probe()函数
29  static int s3c24xx_serial_probe(struct platform_device *dev,
30              struct s3c24xx_uart_info *info)
31  {
32  struct s3c24xx_uart_port *ourport;
33  int ret;
34
35  dbg("s3c24xx_serial_probe(%p, %p) %d\n", dev, info, probe_index);
36
37  ourport = &s3c24xx_serial_ports[probe_index];
38  probe_index++;
39
40  dbg("%s: initialising port %p...\n", __FUNCTION__, ourport);
41    //初始化 UART 端口
42  ret = s3c24xx_serial_init_port(ourport, info, dev);
43  if (ret < 0)
44      goto probe_err;
45
46  dbg("%s: adding port\n", __FUNCTION__);
47  //添加 uart_port
48  uart_add_one_port(&s3c24xx_uart_drv, &ourport->port);
49  platform_set_drvdata(dev, &ourport->port);
50
51  return 0;
52
53   probe_err:
54  return ret;
55  }
56  //42 行调用的 s3c24xx_serial_init_port()函数
57    static  int  s3c24xx_serial_init_port(struct  s3c24xx_uart_port
*ourport,
58                  struct s3c24xx_uart_info *info,
59                  struct platform_device *platdev)
```

```
    60  {
    61  struct uart_port *port = &ourport->port;
    62  struct s3c2410_uartcfg *cfg;
    63  struct resource *res;
    64
    65  dbg("s3c24xx_serial_init_port:  port=%p,  platdev=%p\n",  port,
platdev);
    66
    67  if (platdev == NULL)
    68      return -ENODEV;
    69
    70  cfg = s3c24xx_dev_to_cfg(&platdev->dev);
    71
    72  if (port->mapbase != 0)
    73      return 0;
    74
    75  if (cfg->hwport > 3)
    76      return -EINVAL;
    77
    78  /* 为端口设置 info 成员 */
    79  port->dev   = &platdev->dev;
    80  ourport->info   = info;
    81
    82  /* 初始化 fifosize */
    83  ourport->port.fifosize = info->fifosize;
    84
    85  dbg("s3c24xx_serial_init_port:    %p    (hw    %d)...\n",    port,
cfg->hwport);
    86
    87  port->uartclk = 1;
    88    /* 如果使用流控 */
    89  if (cfg->uart_flags & UPF_CONS_FLOW) {
    90      dbg("s3c24xx_serial_init_port: enabling flow control\n");
    91      port->flags |= UPF_CONS_FLOW;
    92  }
    93
    94  /* 利用平台资源中记录的信息初始化 UART 端口的基地址、中断号 */
    95  res = platform_get_resource(platdev, IORESOURCE_MEM, 0);
    96  if (res == NULL) {
    97      printk(KERN_ERR "failed to find memory resource for uart\n");
```

```
    return -EINVAL;
}

dbg("resource %p (%lx..%lx)\n", res, res->start, res->end);

port->mapbase   = res->start;
port->membase = S3C24XX_VA_UART+(res->start - S3C24XX_PA_UART);
port->irq   = platform_get_irq(platdev, 0);

ourport->clk    = clk_get(&platdev->dev, "uart");

dbg("port: map=%08x, mem=%08x, irq=%d, clock=%ld\n",
    port->mapbase, port->membase, port->irq, port->uartclk);

/* 复位 fifo 并设置 UART */
s3c24xx_serial_resetport(port, cfg);
return 0;
}
//113 行调用的 s3c24xx_serial_resetport()函数
static inline int s3c24xx_serial_resetport(struct uart_port * port,
                  struct s3c2410_uartcfg *cfg)
{
struct s3c24xx_uart_info *info = s3c24xx_port_to_info(port);

return (info->reset_port)(port, cfg);
}
//122 行调用的 info->reset_port()函数
static int s3c2410_serial_resetport(struct uart_port *port,
               struct s3c2410_uartcfg *cfg)
{
dbg("s3c2410_serial_resetport: port=%p (%08lx), cfg=%p\n",
    port, port->mapbase, cfg);

wr_regl(port, S3C2410_UCON,  cfg->ucon);
wr_regl(port, S3C2410_ULCON, cfg->ulcon);

/* 复位两个 fifo */
wr_regl(port,         S3C2410_UFCON,         cfg->ufcon         |
S3C2410_UFCON_RESETBOTH);
wr_regl(port, S3C2410_UFCON, cfg->ufcon);
```

```
137
138 return 0;
139 }
```

在 S3C2410 串口驱动模块被卸载时，它会最终调用 uart_remove_one_port()释放 uart_port，并调用 uart_unregister_driver()注销 uart_driver，如代码清单 14.27 所示。

代码清单 14.27　S3C2410 串口驱动的释放过程

```
1  static void __exit s3c24xx_serial_modexit(void)
2  {
3   s3c2410_serial_exit();
4    //注销 uart_driver
5   uart_unregister_driver(&s3c24xx_uart_drv);
6  }
7
8  static inline void s3c2410_serial_exit(void)
9  {
10  //注销平台驱动
11  platform_driver_unregister(&s3c2410_serial_drv);
12 }
13
14 static int s3c24xx_serial_remove(struct platform_device *dev)
15 {
16  struct uart_port *port = s3c24xx_dev_to_port(&dev->dev);
17
18  //移除 UART 端口
19  if (port)
20      uart_remove_one_port(&s3c24xx_uart_drv, port);
21
22  return 0;
23 }
```

上述代码中对 S3C24xx_serial_remove()的调用发生在 platform_driver_unregister()之后，由于 S3C2410 的 UART 是集成于 SoC 芯片内部的一个独立的硬件单元，因此也被作为一个平台设备而定义。

14.7.4　S3C2410 串口数据收发

S3C2410 串口驱动 uart_ops 结构体的 startup ()成员函数 s3c24xx_serial_startup()用于启动端口，申请端口的发送、接收中断，使能端口的发送和接收，其实现如代码清单 14.28 所示。

代码清单 14.28　S3C2410 串口驱动的 startup ()函数

```
1  static int s3c24xx_serial_startup(struct uart_port *port)
2  {
```

```
3   struct s3c24xx_uart_port *ourport = to_ourport(port);
4   int ret;
5
6   dbg("s3c24xx_serial_startup: port=%p (%08lx,%p)\n",
7       port->mapbase, port->membase);
8
9   rx_enabled(port) = 1;//置接收使能状态为 1
10   //申请接收中断
11  ret = request_irq(RX_IRQ(port),
12            s3c24xx_serial_rx_chars, 0,
13            s3c24xx_serial_portname(port), ourport);
14
15  if (ret != 0) {
16      printk(KERN_ERR "cannot get irq %d\n", RX_IRQ(port));
17      return ret;
18  }
19
20  ourport->rx_claimed = 1;
21
22  dbg("requesting tx irq...\n");
23
24  tx_enabled(port) = 1;//置发送使能状态为 1
25   //申请发送中断
26  ret = request_irq(TX_IRQ(port),
27            s3c24xx_serial_tx_chars, 0,
28            s3c24xx_serial_portname(port), ourport);
29
30  if (ret) {
31      printk(KERN_ERR "cannot get irq %d\n", TX_IRQ(port));
32      goto err;
33  }
34
35  ourport->tx_claimed = 1;
36
37  dbg("s3c24xx_serial_startup ok\n");
38
39  /* 端口复位代码应该已经为端口控制设置了正确的寄存器 */
40
41  return ret;
42
```

```
43  err:
44  s3c24xx_serial_shutdown(port);
45  return ret;
46 }
```

s3c24xx_serial_startup()的“反函数”为 s3c24xx_serial_shutdown()，其释放中断，禁止发送和接收，实现如代码清单 14.29 所示。

代码清单 14.29　S3C2410 串口驱动的 shutdown ()函数

```
1  static void s3c24xx_serial_shutdown(struct uart_port *port)
2  {
3   struct s3c24xx_uart_port *ourport = to_ourport(port);
4
5   if (ourport->tx_claimed) {
6       free_irq(TX_IRQ(port), ourport);
7       tx_enabled(port) = 0; //置发送使能状态为 0
8       ourport->tx_claimed = 0;
9   }
10
11  if (ourport->rx_claimed) {
12      free_irq(RX_IRQ(port), ourport);
13      ourport->rx_claimed = 0;
14      rx_enabled(port) = 0; //置接收使能状态为 1
15  }
16 }
```

S3C2410 串口驱动 uart_ops 结构体的 tx_empty()成员函数 s3c24xx_serial_tx_empty()用于判断发送缓冲区是否为空，其实现如代码清单 14.30 所示，当使能 FIFO 模式的时候，判断 UFSTATn 寄存器，否则判断 UTRSTATn 寄存器的相应位。

代码清单 14.30　S3C2410 串口驱动的 tx_empty()函数

```
1  /* 检查发送缓冲区/FIFO 是否为空 */
2  static unsigned int s3c24xx_serial_tx_empty(struct uart_port *port)
3  {
4    //fifo 模式，检查 UFSTATn 寄存器
5    struct s3c24xx_uart_info *info = s3c24xx_port_to_info(port);
6    unsigned long ufstat = rd_regl(port, S3C2410_UFSTAT);
7    unsigned long ufcon = rd_regl(port, S3C2410_UFCON);
8
9    if (ufcon &S3C2410_UFCON_FIFOMODE)
10   {
11     if ((ufstat &info->tx_fifomask) != 0 ||  //Tx fifo 数据数非 0
12     (ufstat &info->tx_fifofull))   // Tx fifo 满
```

```
13      return 0;   //0: 非空
14
15    return 1; //1: 空
16   }
17
18   return s3c24xx_serial_txempty_nofifo(port);
19 }
20
21 static int s3c24xx_serial_txempty_nofifo(struct uart_port *port)
22 {
23   //非 fifo 模式，检查 UTRSTATn 寄存器
24   return (rd_regl(port, S3C2410_UTRSTAT) &S3C2410_UTRSTAT_TXE);
25 }
```

S3C2410 串口驱动 uart_ops 结构体的 start_tx()成员函数 s3c24xx_serial_start_tx()用于启动发送，而 stop_rx()成员函数 s3c24xx_serial_stop_rx()用于停止发送，代码清单 14.31 所示为这两个函数的实现。

代码清单 14.31　S3C2410 串口驱动 start_tx ()、stop_rx()函数

```
1  static void s3c24xx_serial_start_tx(struct uart_port *port)
2  {
3   if (!tx_enabled(port)) {//如果端口发送未使能
4       if (port->flags & UPF_CONS_FLOW)
5           s3c24xx_serial_rx_disable(port);
6
7       enable_irq(TX_IRQ(port));//使能发送中断
8       tx_enabled(port) = 1;//置端口发送使能状态为 1
9   }
10 }
11
12 static void s3c24xx_serial_stop_tx(struct uart_port *port)
13 {
14  if (tx_enabled(port)) {//如果端口发送已使能
15      disable_irq(TX_IRQ(port));//禁止发送中断
16      tx_enabled(port) = 0;//置端口发送使能状态为 0
17      if (port->flags & UPF_CONS_FLOW)
18          s3c24xx_serial_rx_enable(port);
19  }
20 }
```

S3C2410 串口驱动 uart_ops 结构体的 stop_rx ()成员函数 s3c24xx_serial_stop_rx ()用于停止接收，代码清单 14.32 所示为这个函数的实现。注意 uart_ops 中没有 start_rx()

成员，因为接收并不是由“我方”启动，而是由“它方”的发送触发“我方”的接收中断，“我方”再被动响应。

代码清单 14.32　S3C2410 串口驱动的 stop_rx ()函数

```
1  static void s3c24xx_serial_stop_rx(struct uart_port *port)
2  {
3   if (rx_enabled(port)) {//如果接收为使能
4       dbg("s3c24xx_serial_stop_rx: port=%p\n", port);
5       disable_irq(RX_IRQ(port));//禁止接收中断
6       rx_enabled(port) = 0;//置接收使能状态为 0
7   }
8  }
```

在 S3C2410 串口驱动中，与数据收发关系最密切的函数不是上述的 uart_ops 成员函数，而是 s3c24xx_serial_startup() 为发送和接收中断注册的中断处理函数 s3c24xx_serial_rx_chars()和 s3c24xx_ serial_tx_chars()。

s3c24xx_serial_rx_chars()读取 URXHn 寄存器以获得接收到的字符，并调用 uart_insert_char()将该字符添加到 tty 设备的 flip 缓冲区中，当接收到 64 个字符或者不再能接收字符后，调用 tty_flip_ buffer_push()函数向上层“推”tty 设备的 flip 缓冲，其实现如代码清单 14.33 所示。

代码清单 14.33　S3C2410 串口驱动的接收中断处理函数

```
1  static irqreturn_t s3c24xx_serial_rx_chars(int irq, void *dev_id,
struct
2    pt_regs *regs)
3  {
4    struct s3c24xx_uart_port *ourport = dev_id;
5    struct uart_port *port = &ourport->port;   //获得 uart_port
6    struct tty_struct *tty = port->info->tty; //获得 tty_struct
7    unsigned int ufcon, ch, flag, ufstat, uerstat;
8    int max_count = 64;
9
10   while (max_count-- > 0)
11   {
12     ufcon = rd_regl(port, S3C2410_UFCON);
13     ufstat = rd_regl(port, S3C2410_UFSTAT);
14     //如果接收到 0 个字符
15     if (s3c24xx_serial_rx_fifocnt(ourport, ufstat) == 0)
16       break;
17
18     uerstat = rd_regl(port, S3C2410_UERSTAT);
19     ch = rd_regb(port, S3C2410_URXH); //读出字符
```

```
20
21    if (port->flags &UPF_CONS_FLOW)
22    {
23      int txe = s3c24xx_serial_txempty_nofifo(port);
24
25      if (rx_enabled(port)) //如果端口为使能接收状态
26      {
27        if (!txe) //如果发送缓冲区为空
28        {
29          rx_enabled(port) = 0; //置端口使能接收状态为 0
30          continue;
31        }
32      }
33      else   //端口为禁止接收状态
34      {
35        if (txe)  //如果发送缓冲区非空
36        {
37          ufcon |= S3C2410_UFCON_RESETRX;
38          wr_regl(port, S3C2410_UFCON, ufcon);
39          rx_enabled(port) = 1;//置端口使能接收状态为 1
40          goto out;
41        }
42        continue;
43      }
44    }
45
46    /* 将接收到的字符写入 buffer */
47    flag = TTY_NORMAL;
48    port->icount.rx++;
49
50    if (unlikely(uerstat &S3C2410_UERSTAT_ANY))
51    {
52      dbg("rxerr: port ch=0x%02x, rxs=0x%08x\n", ch, uerstat);
53
54       if (uerstat &S3C2410_UERSTAT_BREAK)
55      {
56        dbg("break!\n");
57        port->icount.brk++;
58        if (uart_handle_break(port))
59          goto ignore_char;
```

```
    60      }
    61
    62      if (uerstat &S3C2410_UERSTAT_FRAME)
    63        port->icount.frame++;
    64      if (uerstat &S3C2410_UERSTAT_OVERRUN)
    65        port->icount.overrun++;
    66
    67      uerstat &= port->read_status_mask;
    68
    69      if (uerstat &S3C2410_UERSTAT_BREAK)
    70        flag = TTY_BREAK;
    71      else if (uerstat &S3C2410_UERSTAT_PARITY)
    72        flag = TTY_PARITY;
    73                  else   if   (uerstat   &(S3C2410_UERSTAT_FRAME   |
S3C2410_UERSTAT_OVERRUN))
    74        flag = TTY_FRAME;
    75    }
    76
    77    if (uart_handle_sysrq_char(port, ch, regs)) //处理 sysrq 字符
    78      goto ignore_char;
    79    //插入字符到 tty 设备的 flip 缓冲
    80     uart_insert_char(port, uerstat, S3C2410_UERSTAT_OVERRUN, ch,
flag);
    81
    82    ignore_char: continue;
    83  }
    84  tty_flip_buffer_push(tty);  //刷新 tty 设备的 flip 缓冲
    85
    86  out: return IRQ_HANDLED;
    87 }
```

上述代码第 80 行的 uart_insert_char()函数是串口核心层对 tty_insert_flip_char()的封装，它作为内联函数被定义于 serial_core.h 文件中。

如代码清单 14.34 所示，s3c24xx_serial_tx_chars()读取 uart_info 中环形缓冲区中的字符，写入调用 UTXHn 寄存器。

代码清单 14.34　S3C2410 串口驱动发送中断处理函数

```
    1  static irqreturn_t s3c24xx_serial_tx_chars(int irq, void *id, struct
pt_regs
    2    *regs)
    3  {
    4    struct s3c24xx_uart_port *ourport = id;
```

```
5   struct uart_port *port = &ourport->port;
6   struct circ_buf *xmit = &port->info->xmit;  //得到环形缓冲区
7   int count = 256;   //最多一次发 256 个字符
8
9   if (port->x_char) //如果定义了 xchar，发送
10  {
11    wr_regb(port, S3C2410_UTXH, port->x_char);
12    port->icount.tx++;
13    port->x_char = 0;
14    goto out;
15  }
16
17  /* 如果没有更多的字符需要发送，或者 UART Tx 停止，则停止 UART 并退出 */
18  if (uart_circ_empty(xmit) || uart_tx_stopped(port))
19  {
20    s3c24xx_serial_stop_tx(port);
21    goto out;
22  }
23
24  /* 尝试把环行 buffer 中的数据发空 */
25  while (!uart_circ_empty(xmit) && count-- > 0)
26  {
27    if (rd_regl(port, S3C2410_UFSTAT) &ourport->info->tx_fifofull)
28      break;
29
30    wr_regb(port, S3C2410_UTXH, xmit->buf[xmit->tail]);
31    xmit->tail = (xmit->tail + 1) &(UART_XMIT_SIZE - 1);
32    port->icount.tx++;
33  }
34  /* 如果环形缓冲区中剩余的字符少于 WAKEUP_CHARS，唤醒上层 */
35  if (uart_circ_chars_pending(xmit) < WAKEUP_CHARS)
36    uart_write_wakeup(port);
37
38  if (uart_circ_empty(xmit)) //如果发送环形 buffer 为空
39    s3c24xx_serial_stop_tx(port); //停止发送
40
41  out: return IRQ_HANDLED;
42 }
```

上述代码第 35 行的宏 WAKEUP_CHARS 的含义为：当发送环形缓冲区中的字符

数小于该数时，驱动将请求上层向下传递更多的数据，uart_write_wakeup()完成此目的。

uart_circ_chars_pending()、uart_circ_empty()是定义于 serial_core.h 中的宏，分别返回环形缓冲区剩余的字符数以及判断缓冲区是否为空。

14.7.5　S3C2410 串口线路设置

S3C2410 串口驱动 uart_ops 结构体的 set_termios()成员函数用于改变端口的参数设置，包括波特率、字长、停止位、奇偶校验等，它会根据传递给它的 port、termios 参数成员的值设置 S3C2410 UART 的 ULCONn、UCONn、UMCONn 等寄存器，其实现如代码清单 14.35 所示。

代码清单 14.35　S3C2410 串口驱动 set_termios()函数

```
1  static void s3c24xx_serial_set_termios(struct uart_port *port,
2                          struct termios *termios,
3                          struct termios *old)
4  {
5   struct s3c2410_uartcfg *cfg = s3c24xx_port_to_cfg(port);
6   struct s3c24xx_uart_port *ourport = to_ourport(port);
7   struct s3c24xx_uart_clksrc *clksrc = NULL;
8   struct clk *clk = NULL;
9   unsigned long flags;
10  unsigned int baud, quot;
11  unsigned int ulcon;
12  unsigned int umcon;
13
14  /* 不支持 modem 控制信号线 */
15  termios->c_cflag &= ~(HUPCL | CMSPAR);
16  termios->c_cflag |= CLOCAL;
17
18  /* 请求内核计算分频以便产生对应的波特率 */
19  baud = uart_get_baud_rate(port, termios, old, 0, 115200*8);
20
21  if (baud == 38400 && (port->flags & UPF_SPD_MASK) == UPF_SPD_CUST)
22      quot = port->custom_divisor;
23  else
24      quot = s3c24xx_serial_getclk(port, &clksrc, &clk, baud);
25
26  /* 检查以确定是否需要改变时钟源 */
27  if (ourport->clksrc != clksrc || ourport->baudclk != clk) {
28      s3c24xx_serial_setsource(port, clksrc);
29
30      if (ourport->baudclk != NULL && !IS_ERR(ourport->baudclk)) {
```

```
31          clk_disable(ourport->baudclk);
32          ourport->baudclk  = NULL;
33      }
34
35      clk_enable(clk);
36
37      ourport->clksrc = clksrc;
38      ourport->baudclk = clk;
39  }
40
41    /* 设置字长 */
42  switch (termios->c_cflag & CSIZE) {
43  case CS5:
44      dbg("config: 5bits/char\n");
45      ulcon = S3C2410_LCON_CS5;
46      break;
47  case CS6:
48      dbg("config: 6bits/char\n");
49      ulcon = S3C2410_LCON_CS6;
50      break;
51  case CS7:
52      dbg("config: 7bits/char\n");
53      ulcon = S3C2410_LCON_CS7;
54      break;
55  case CS8:
56  default:
57      dbg("config: 8bits/char\n");
58      ulcon = S3C2410_LCON_CS8;
59      break;
60  }
61
62  /* 保留以前的 lcon IR 设置 */
63  ulcon |= (cfg->ulcon & S3C2410_LCON_IRM);
64
65  if (termios->c_cflag & CSTOPB)
66      ulcon |= S3C2410_LCON_STOPB;
67      /* 设置是否采用 RTS、CTS 自动流空 */
68  umcon = (termios->c_cflag & CRTSCTS) ? S3C2410_UMCOM_AFC : 0;
69
70  if (termios->c_cflag & PARENB) {
```

```
    71      if (termios->c_cflag & PARODD)
    72          ulcon |= S3C2410_LCON_PODD;//奇校验
    73      else
    74          ulcon |= S3C2410_LCON_PEVEN;//偶校验
    75  } else {
    76      ulcon |= S3C2410_LCON_PNONE;//无校验
    77  }
    78
    79  spin_lock_irqsave(&port->lock, flags);
    80
    81  dbg("setting ulcon to %08x, brddiv to %d\n", ulcon, quot);
    82
    83  wr_regl(port, S3C2410_ULCON, ulcon);
    84  wr_regl(port, S3C2410_UBRDIV, quot);
    85  wr_regl(port, S3C2410_UMCON, umcon);
    86
    87  dbg("uart: ulcon = 0x%08x, ucon = 0x%08x, ufcon = 0x%08x\n",
    88     rd_regl(port, S3C2410_ULCON),
    89     rd_regl(port, S3C2410_UCON),
    90     rd_regl(port, S3C2410_UFCON));
    91
    92  /* 更新端口的超时 */
    93  uart_update_timeout(port, termios->c_cflag, baud);
    94
    95  /*对什么字符状态标志感兴趣? */
    96  port->read_status_mask = S3C2410_UERSTAT_OVERRUN;
    97  if (termios->c_iflag & INPCK)
    98      port->read_status_mask     |=     S3C2410_UERSTAT_FRAME     |
S3C2410_UERSTAT_PARITY;
    99
   100 /* 要忽略什么字符状态标志? */
   101 port->ignore_status_mask = 0;
   102 if (termios->c_iflag & IGNPAR)
   103     port->ignore_status_mask |= S3C2410_UERSTAT_OVERRUN;
   104 if (termios->c_iflag & IGNBRK && termios->c_iflag & IGNPAR)
   105     port->ignore_status_mask |= S3C2410_UERSTAT_FRAME;
   106
   107 /* 如果 CREAD 未设置，忽略所用字符 */
   108 if ((termios->c_cflag & CREAD) == 0)
   109     port->ignore_status_mask |= RXSTAT_DUMMY_READ;
```

```
    110
    111 spin_unlock_irqrestore(&port->lock, flags);
    112 }
```

由于 S3C2410 集成 UART 并不包含完整的 Modem 控制信号线，因此其 uart_ops 结构体的 get_mctrl()、set_mctrl()成员函数的实现非常简单，如代码清单 14.36 所示，get_mctrl()返回DSR一直有效，而CTS则根据UMSTATn寄存器的内容获得，set_mctrl()目前为空。

代码清单 14.36　S3C2410 串口驱动的 get_mctrl()和 set_mctrl()函数

```
    1   static unsigned int s3c24xx_serial_get_mctrl(struct uart_port
*port)
    2 {
    3   unsigned int umstat = rd_regb(port,S3C2410_UMSTAT);
    4
    5   if (umstat & S3C2410_UMSTAT_CTS)//CTS信号有效（低电平）
    6       return TIOCM_CAR | TIOCM_DSR | TIOCM_CTS;
    7   else
    8       return TIOCM_CAR | TIOCM_DSR;
    9 }
    10
    11  static void s3c24xx_serial_set_mctrl(struct uart_port *port,
unsigned int mctrl)
    12 {
    13  /* todo: 可能移除 AFC，并手工设置 CTS */
    14 }
```

14.8 总结

TTY 设备驱动的主体工作围绕 tty_driver 这个结构体的成员函数展开，主要应实现其中的数据发送和接收流程以及 tty 设备线路设置接口函数。

针对串口，内核实现了串口核心层，这个层实现了串口设备通用的 tty_driver。因此，串口设备驱动的主体工作从 tty_driver 转移到了 uart_driver。