








第7章 字符设备驱动程序(char driver)

7.1 概述

在 linux 0.11 内核中, 字符设备主要包括控制终端设备和串行终端设备。本章的代码就是用于对这些设备的输入输出进行操作。有关终端驱动程序的工作原理可参考 M.J.Bach 的《UNIX 操作系统设计》第 10 章第 3 节内容。

列表 7-1 linux/kernel/chr_drv 目录

	文件名	大小	最后修改时间(GMT)	说明
	Makefile	2443 bytes	1991-12-02 03:21:41	Make 配置文件
	console.c	14568 bytes	1991-11-23 18:41:21	终端处理程序
	keyboard.S	12780 bytes	1991-12-04 15:07:58	键盘中断处理程序
	rs_io.s	2718 bytes	1991-10-02 14:16:30	串行线路处理程序
	serial.c	1406 bytes	1991-11-17 21:49:05	串行终端处理程序
	tty_io.c	7634 bytes	1991-12-08 18:09:15	终端 IO 处理程序
	tty_ioctl.c	4979 bytes	1991-11-25 19:59:38	终端 IO 控制程序

7.2 总体功能描述

本章的程序可分成三块。一块是关于 RS-232 串行线路驱动程序, 包括程序 rs_io.s 和 serial.c; 另一块是涉及控制台驱动程序, 这包括键盘中断驱动程序 keyboard.S 和控制台显示驱动程序 console.c; 第三部分是终端驱动程序与上层接口部分, 包括终端输入输出程序 tty_io.c 和终端控制程序 tty_ioctl.c。下面我们首先概述终端控制驱动程序实现的基本原理, 然后再分这三部分分别说明它们的基本功能。

7.2.1 终端驱动程序基本原理

终端驱动程序用于控制终端设备, 在终端设备和进程之间传输数据, 并对所传输的数据进行一定的处理。用户在键盘上键入的原始数据 (Raw data), 在通过终端程序处理后, 被传送给一个接收进程; 而进程向终端发送的数据, 在终端程序处理后, 被显示在终端屏幕上或者通过串行线路被发送到远程终端。根据终端程序对待输入或输出数据的方式, 可以把终端工作模式分成两种。一种是规范模式 (canonical), 此时经过终端程序的数据将被进行变换处理, 然后再送出。例如把 TAB 字符扩展为 8 个空格字符, 用键入的删除字符 (backspace) 控制删除前面键入的字符等。使用的处理函数一般称为行规则(line discipline)模块。另一种是非规范模式或称原始(raw)模式。在这种模式下, 行规则程序仅在终端与进程之间传送数据, 而不对数据进行规范模式的变换处理。

在终端驱动程序中, 根据它们与设备的关系, 以及在执行流程中的位置, 可以分为字符设备的直接驱动程序和与上层直接联系的接口程序。我们可以用图 7-1 示意图来表示这种控制关系。

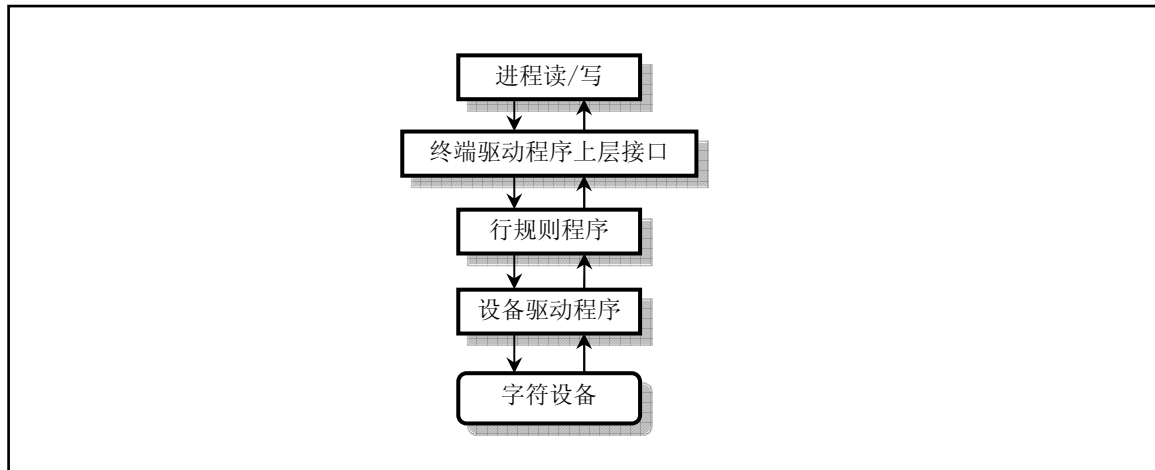


图 7-1 终端驱动程序控制流程

7.2.2 终端基本数据结构

每个终端设备都对应有一个 `tty_struct` 数据结构，主要用来保存终端设备当前参数设置、所属的前台进程组 ID 和字符 IO 缓冲队列等信息。该结构定义在 `include/linux/tty.h` 文件中，其结构如下所示：

```
struct tty_struct {
    struct termios termios;           // 终端 io 属性和控制字符数据结构。
    int pgrp;                          // 所属进程组。
    int stopped;                       // 停止标志。
    void (*write)(struct tty_struct * tty); // tty 写函数指针。
    struct tty_queue read_q;          // tty 读队列。
    struct tty_queue write_q;         // tty 写队列。
    struct tty_queue secondary;       // tty 辅助队列(存放规范模式字符序列)，
};                                       // 可称为规范(熟)模式队列。
extern struct tty_struct tty_table[]; // tty 结构数组。
```

Linux 内核使用了数组 `tty_table[]` 来保存系统中每个终端设备的信息。每个数组项是一个数据结构 `tty_struct`，对应系统中一个终端设备。Linux 0.11 内核共支持三个终端设备。一个是控制台设备，另外两个是使用系统上两个串行端口的串行终端设备。

`termios` 结构用于存放对应终端设备的 io 属性。有关该结构的详细描述见下面说明。`pgrp` 是进程组标识，它指明一个会话中处于前台的进程组，即当前拥有该终端设备的进程组。`pgrp` 主要用于进程的作业控制操作。`stopped` 是一个标志，表示对应终端设备是否已经停止使用。函数指针 `*write()` 是该终端设备的输出处理函数，对于控制台终端，它负责驱动显示硬件，在屏幕上显示字符等信息。对于通过系统串行端口连接的串行终端，它负责把输出字符发送到串行端口。

终端所处理的数据被保存在 3 个 `tty_queue` 结构的字符缓冲队列中（或称为字符表），见下面所示：

```
struct tty_queue {
    unsigned long data;                // 等待队列缓冲区中当前数据统计值。
                                         // 对于串口终端，则存放串口端口地址。
    unsigned long head;                // 缓冲区中数据头指针。
    unsigned long tail;                // 缓冲区中数据尾指针。
    struct task_struct * proc_list;   // 等待本缓冲队列的进程列表。
```

```
char buf[1024];           // 队列的缓冲区。
};
```

每个字符缓冲队列的长度是 1K 字节。其中读缓冲队列 `read_q` 用于临时存放从键盘或串行终端输入的原始 (raw) 字符序列；写缓冲队列 `write_q` 用于存放写到控制台显示屏或串行终端去的数据；根据 ICANON 标志，辅助队列 `secondary` 用于存放从 `read_q` 中取出的经过行规则程序处理（过滤）过的数据，或称为熟(cooked)模式数据。这是在行规则程序把原始数据中的特殊字符如删除 (backspace) 字符变换后的规范输入数据，以字符行为单位供应用程序读取使用。上层终端读函数 `tty_read()` 即用于读取 `secondary` 队列中的字符。

在读入用户键入的数据时，中断处理汇编程序只负责把原始字符数据放入输入缓冲队列中，而由中断处理过程中调用的 C 函数 (`copy_to_cooked()`) 来处理字符的变换工作。例如当进程向一个终端写数据时，终端驱动程序就会调用行规则函数 `copy_to_cooked()`，把用户缓冲区中的所有数据数据到写缓冲队列中，并将数据发送到终端上显示。在终端上按下一个键时，所引发的键盘中断处理过程会把按键扫描码对应的字符放入读队列 `read_q` 中，并调用规范模式处理程序把 `read_q` 中的字符经过处理再放入辅助队列 `secondary` 中。与此同时，如果终端设备设置了回显标志 (L_ECHO)，则也把该字符放入写队列 `write_q` 中，并调用终端写函数把该字符显示在屏幕上。通常除了象键入密码或其它特殊要求以外，回显标志都是置位的。我们可以通过修改终端的 `termios` 结构中的信息来改变这些标志值。

在上述 `tty_struct` 结构中还包括一个 `termios` 结构，该结构定义在 `include/termios.h` 头文件中，其字段内容如下所示：

```
struct termios {
    unsigned long c_iflag;      /* input mode flags */    // 输入模式标志。
    unsigned long c_oflag;      /* output mode flags */   // 输出模式标志。
    unsigned long c_cflag;      /* control mode flags */  // 控制模式标志。
    unsigned long c_lflag;      /* local mode flags */    // 本地模式标志。
    unsigned char c_line;        /* line discipline */     // 线路规程 (速率)。
    unsigned char c_cc[NCCS];    /* control characters */  // 控制字符数组。
};
```

其中，`c_iflag` 是输入模式标志集。Linux 0.11 内核实现了 POSIX.1 定义的所有 11 个输入标志，参见 `termios.h` 头文件中的说明。终端设备驱动程序用这些标志来控制如何对终端输入的字符进行变换（过滤）处理。例如是否需要把输入的的换行符 (NL) 转换成回车符 (CR)、是否需要把输入的大写字符转换成小写字符（因为以前有些终端设备只能输入大写字符）等。在 Linux 0.11 内核中，相关的处理函数是 `tty_io.c` 文件中的 `copy_to_cooked()`。

`c_oflag` 是输出模式标志集。终端设备驱动程序使用这些标志控制如何把字符输出到终端上。`c_cflag` 是控制模式标志集。主要用于定义串行终端传输特性，包括波特率、字符比特位数以及停止位数等。`c_lflag` 是本地模式标志集。主要用于控制驱动程序与用户的交互。例如是否需要回显 (Echo) 字符、是否需要把擦除字符直接显示在屏幕上、是否需要让终端上键入的控制字符产生信号。这些操作也同样在 `copy_to_cooked()` 函数中实现。

上述 4 种标志集的类型都是 `unsigned long`，每个比特位可表示一种标志，因此每个标志集最多可有 32 个输入标志。所有这些标志及其含义可参见 `termios.h` 头文件。

`c_cc[]` 数组包含了所有可以修改的特殊字符。例如你可以通过修改其中的中断字符 (^C) 由其它按键产生。其中 NCCS 是数组的长度值。

因此，利用系统调用 `ioctl` 或使用相关函数 (`tcsetattr()`)，我们可以通过修改 `termios` 结构中的信息来改变终端的设置参数。行规则函数即是根据这些设置参数进行操作。例如，控制终端是否要对键入的字符

进行回显、设置串行终端传输的波特率、清空读缓冲队列和写缓冲队列。

当用户修改终端参数，将规范模式标志复位，则就会把终端设置为工作在原始模式，此时行规则程序会把用户键入的数据原封不动地传送给用户，而回车符也被当作普通字符处理。因此，在用户使用系统调用 `read` 时，就应该作出某种决策方案以判断系统调用 `read` 什么是否算完成并返回。这将由终端 `termios` 结构中的 `VTIME` 和 `VMIN` 控制字符决定。这两个是读操作的超时定时值。`VMIN` 表示为了满足读操作，需要读取的最少字符数；`VTIME` 则是一个读操作等待定时值。

我们可以使用命令 `stty` 来查看当前终端设备 `termios` 结构中标志的设置情况。在 Linux 0.1x 系统命令行提示符下键入 `stty` 命令会显示以下信息：

```
[/root]# stty
-----Characters-----
INTR:  '^C'  QUIT:  '^\'  ERASE:  '^H'  KILL:  '^U'  EOF:  '^D'
TIME:   0    MIN:   1    SWTC:  '^@'  START:  '^Q'  STOP:  '^S'
SUSP:  '^Z'  EOL:   '^@'  EOL2:  '^@'  LNEXT:  '^V'
DISCARD: '^O'  REPRINT: '^R'  RWERASE: '^W'
-----Control Flags-----
-CSTOPB  CREAD -PARENB -PARODD  HUPCL -CLOCAL -CRTSCTS
Baud rate: 9600 Bits: CS8
-----Input Flags-----
-IGNBRK -BRKINT -IGNPAR -PARMRK -INPCK -ISTRIP -INLCR -IGNCR
ICRNL -IUCLC  IXON  -IXANY  IXOFF -IMAXBEL
-----Output Flags-----
OPOST -OLCUC  ONLCR -OCRNL -ONOCR -ONLRET -OFILL -OFDEL
Delay modes: CR0 NL0 TAB0 BS0 FF0 VT0
-----Local Flags-----
ISIG ICANON -XCASE  ECHO -ECHOE -ECHOK -ECHONL -NOFLSH
-TOSTOP ECHOCTL ECHOPRT ECHOKE -FLUSHO -PENDIN -IEXTEN
rows 0 cols 0
```

其中带有减号标志表示没有设置。另外对于现在的 Linux 系统，需要键入 `'stty -a'` 才能显示所有这些信息，并且显示格式有所区别。

终端程序所使用的上述主要数据结构和它们之间的关系可见图 7-2 所示。

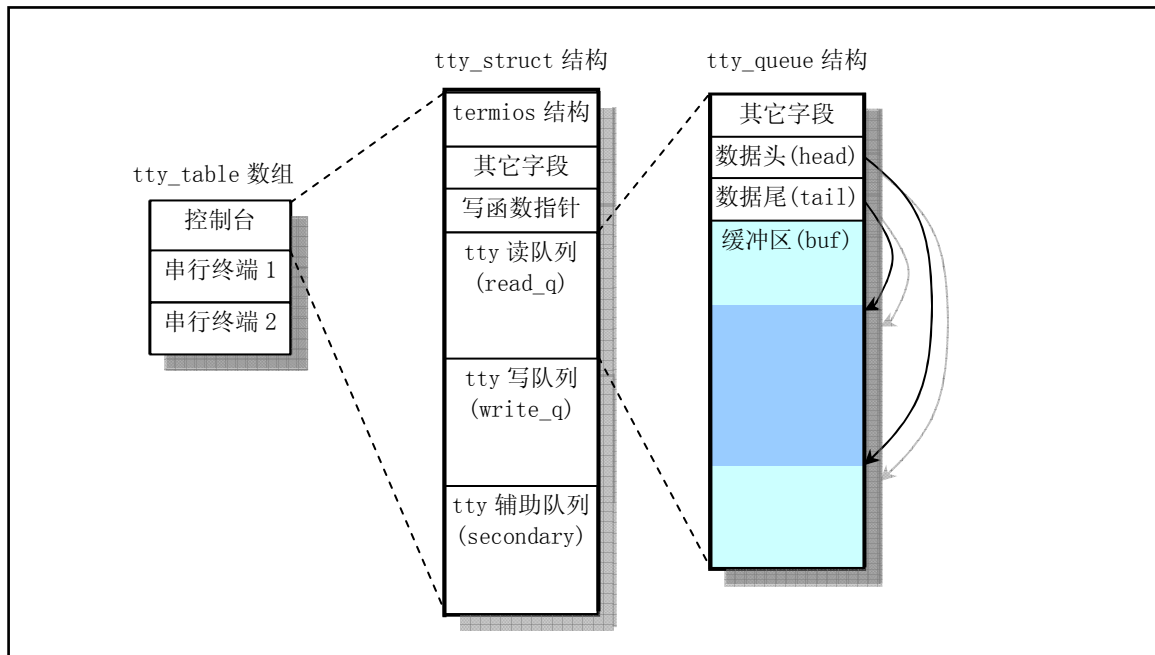


图 7-2 终端程序的数据结构

7.2.3 规范模式和非规范模式

7.2.3.1 规范模式

当 `c_lflag` 中的 `ICANON` 标志置位时，则按照规范模式对终端输入数据进行处理。此时输入字符被装配成行，进程以字符行的形式读取。当一行字符输入后，终端驱动程序会立刻返回。行的定界符有 `NL`、`EOL`、`EOL2` 和 `EOF`。其中除最后一个 `EOF`（文件结束）将被处理程序删除外，其余四个字符将被作为一行的最后一个字符返回给调用程序。

在规范模式下，终端输入的以下字符将被处理：`ERASE`、`KILL`、`EOF`、`EOL`、`REPRINT`、`WERASE` 和 `EOL2`。

`ERASE` 是擦除字符（Backspace）。在规范模式下，当 `copy_to_cooked()` 函数遇该输入字符时会删除缓冲队列中最后输入的一个字符。若队列中最后一个字符是上一行的字符（例如是 `NL`），则不作任何处理。此后该字符被忽略，不放到缓冲队列中。

`KILL` 是删行字符。它删除队列中最后一行字符。此后该字符被忽略掉。

`EOF` 是文件结束符。在 `copy_to_cooked()` 函数中该字符以及行结束字符 `EOL` 和 `EOL2` 都将被当作回车符来处理。在读操作函数中遇到该字符将立即返回。`EOF` 字符不会放入队列中而是被忽略掉。

`REPRINT` 和 `WERASE` 是扩展规范模式下识别的字符。`REPRINT` 会让所有未读的输入被输出。而 `WERASE` 用于擦除单词（跳过空白字符）。在 Linux 0.11 中，程序忽略了对这两个字符的识别和处理。

7.2.3.2 非规范模式

如果 `ICANON` 处于复位状态，则终端程序工作在非规范模式下。此时终端程序不对上述字符进行处理，而是将它们当作普通字符处理。输入数据也没有行的概念。终端程序何时返回读进程是由 `MIN` 和 `TIME` 的值确定。这两个变量是 `c_cc[]` 数组中的变量。通过修改它们即可改变在非规范模式下进程读字符的处理方式。

`MIN` 指明读操作最少需要读取的字符数；`TIME` 指定等待读取字符的超时值（计量单位是 1/10 秒）。根据它们的值可分四种情况来说明。

1. `MIN>0`, `TIME>0`

此时 `TIME` 是一个字符间隔超时定时值，在接收到第一个字符后才起作用。在超时之前，若先

接收到了 MIN 个字符，则读操作立刻返回。若在收到 MIN 个字符之前超时了，则读操作返回已经接收到的字符数。此时起码能返回一个字符。因此在接收到一个字符之前若 secondary 空，则读进程将被阻塞（睡眠）。

2. MIN>0, TIME=0

此时只有在收到 MIN 个字符时读操作才返回。否则就无限期等待（阻塞）。

3. MIN=0, TIME>0

此时 TIME 是一个读操作超时定时值。当收到一个字符或者已超时，则读操作就立刻返回。如果是超时返回，则读操作返回 0 个字符。

4. MIN=0, TIME=0

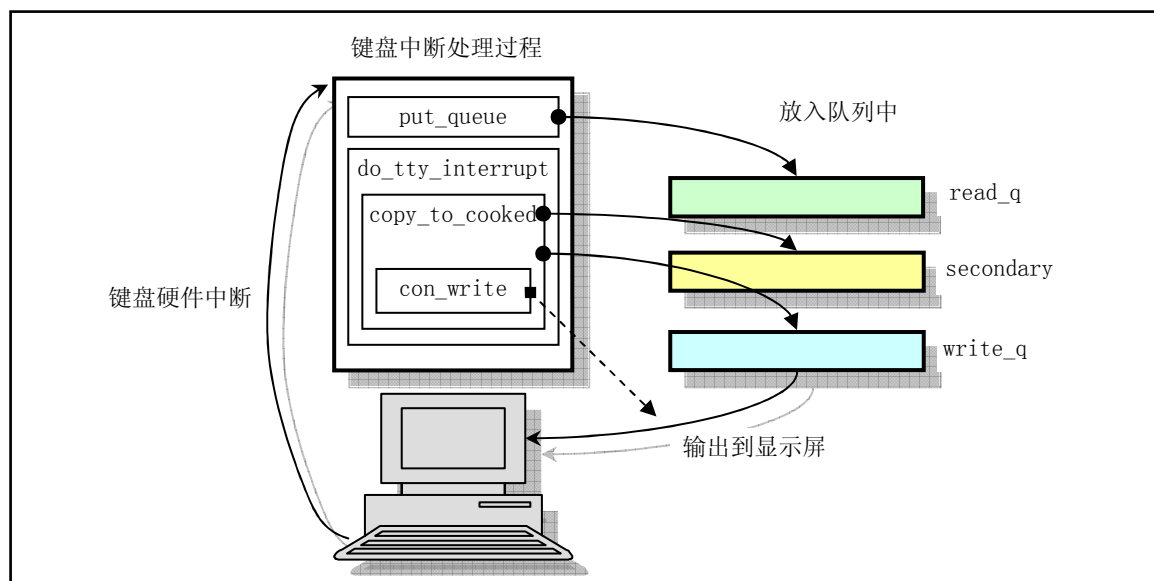
在这种设置下，如果队列中有数据可以读取，则读操作读取需要的字符数。否则立刻返回 0 个字符数。

在以上四中情况中，MIN 仅表明最少读到的字符数。如果进程要求读取比 MIN 要多的字符，那么只要队列中有就可能满足进程的当前需求。有关对终端设备的读操作处理，请参见程序 tty_io.c 中的 tty_read() 函数。

7.2.4 控制台驱动程序

在 Linux 0.11 内核中，终端控制台驱动程序涉及 keyboard.S 和 console.c 程序。keyboard.S 用于处理用户键入的字符，把它们放入读缓冲队列 read_q 中，并调用 copy_to_cooked() 函数读取 read_q 中的字符，经转换后放入辅助缓冲队列 secondary。console.c 程序实现控制台终端的输出处理。

例如，当用户在键盘上键入了一个字符时，会引起键盘中断响应（中断请求信号 IRQ1, 对应中断号 INT 33），此时键盘中断处理程序就会从键盘控制器读入对应的键盘扫描码，然后根据使用的键盘扫描码映射表译成相应字符，放入 tty 读队列 read_q 中。然后调用中断处理程序的 C 函数 do_tty_interrupt()，它又直接调用行规则函数 copy_to_cooked() 对该字符进行过滤处理，并放入 tty 辅助队列 secondary 中，同时将该字符放入 tty 写队列 write_q 中，并调用写控制台函数 con_write()。此时如果该终端的回显（echo）属性是设置的，则该字符会显示到屏幕上。do_tty_interrupt() 和 copy_to_cooked() 函数在 tty_io.c 中实现。整个操作过程见图 7-3 所示。



对于进程执行 `tty` 写操作，终端驱动程序是一个字符一个字符进行处理的。在写缓冲队列 `write_q` 没有满时，就从用户缓冲区取一个字符，经过处理放入 `write_q` 中。当把用户数据全部放入 `write_q` 队列或者此时 `write_q` 已满，就调用终端结构 `tty_struct` 中指定的写函数，把 `write_q` 缓冲队列中的数据输出到控制台。对于控制台终端，其写函数是 `con_write()`，在 `console.c` 程序中实现。

有关控制台终端操作的驱动程序，主要涉及两个程序。一个是键盘中断处理程序 `keyboard.S`，主要用于把用户键入的字符并放入 `read_q` 缓冲队列中；另一个是屏幕显示处理程序 `console.c`，用于从 `write_q` 队列中取出字符并显示在屏幕上。所有这三个字符缓冲队列与上述函数或文件的关系都可以用图 7-4 清晰地表示出来。

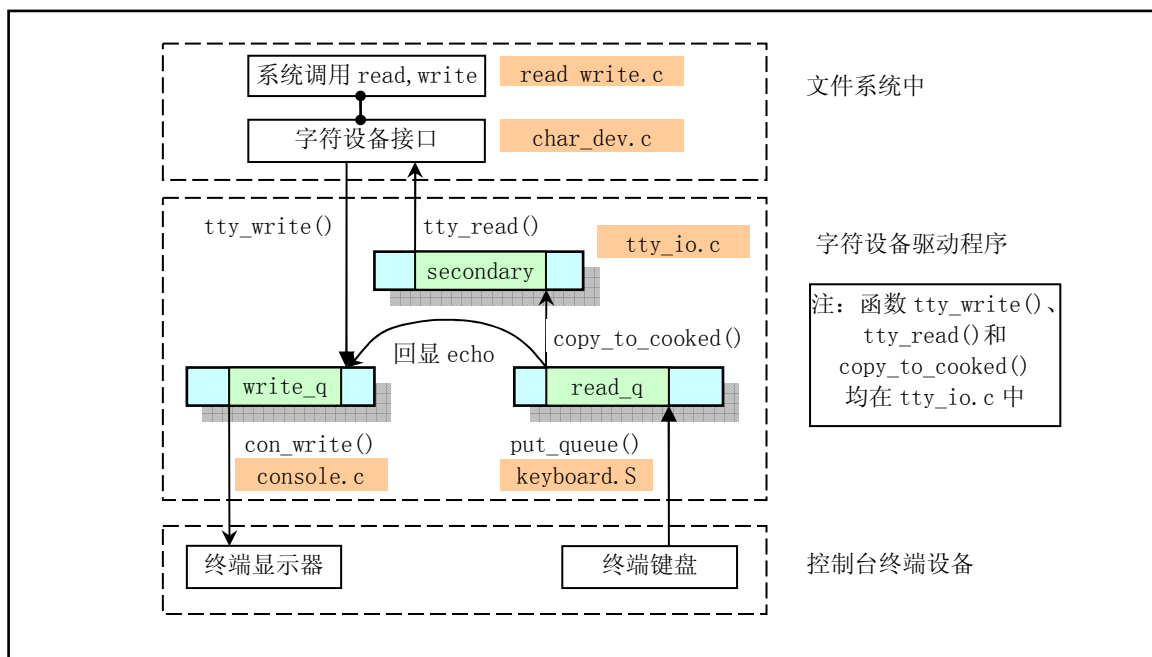


图 7-4 控制台终端字符缓冲队列以及函数和程序之间的关系

7.2.5 串行终端驱动程序

处理串行终端操作的程序有 `serial.c` 和 `rs_io.s`。`serial.c` 程序负责对串行端口进行初始化操作。另外，通过取消对发送保持寄存器空中断允许的屏蔽来开启串行中断发送字符操作。`rs_io.s` 程序是串行中断处理过程。主要根据引发中断的 4 种原因分别进行处理。

引起系统发生串行中断的情况有：a. 由于 `modem` 状态发生了变化；b. 由于线路状态发生了变化；c. 由于接收到字符；d. 由于在中断允许标志寄存器中设置了发送保持寄存器中断允许标志，需要发送字符。对引起中断的前两种情况的处理过程是通过读取对应状态寄存器值，从而使其复位。对于由于接收到字符的情况，程序首先把该字符放入读缓冲队列 `read_q` 中，然后调用 `copy_to_cooked()` 函数转换成以字符行为单位的规范模式字符放入辅助队列 `secondary` 中。对于需要发送字符的情况，则程序首先从写缓冲队列 `write_q` 尾指针处取出一个字符发送出去，再判断写缓冲队列是否已空，若还有字符则循环执行发送操作。

对于通过系统串行端口接入的终端，除了需要与控制台类似的处理外，还需要进行串行通信的输入/输出处理操作。数据的读入是由串行中断处理程序放入读队列 `read_q` 中，随后执行与控制台终端一样的操作。

例如，对于一个接在串行端口 1 上的终端，键入的字符将首先通过串行线路传送到主机，引起主机串行口 1 中断请求。此时串行口中断处理程序就会将字符放入串行终端 1 的 `tty` 读队列 `read_q` 中，然后

调用中断处理程序的 C 函数 `do_tty_interrupt()`，它又直接调用行规则函数 `copy_to_cooked()` 对该字符进行过滤处理，并放入 tty 辅助队列 `secondary` 中，同时把该字符放入 tty 写队列 `write_q` 中，并调用写串行终端 1 的函数 `rs_write()`。该函数又会把字符回送给串行终端，此时如果该终端的回显 (echo) 属性是设置的，则该字符会显示在串行终端的屏幕上。

当进程需要写数据到一个串行终端上时，操作过程与写终端类似，只是此时终端的 `tty_struct` 数据结构中的写函数是串行终端写函数 `rs_write()`。该函数取消对发送保持寄存器空允许中断的屏蔽，从而在发送保持寄存器为空时就会引起串行中断发生。而该串行中断过程则根据此次引起中断的原因，从 `write_q` 写缓冲队列中取出一个字符并放入发送保持寄存器中进行字符发送操作。该操作过程也是一次中断发送一个字符，到最后 `write_q` 为空时就会再次屏蔽发送保持寄存器空允许中断位，从而禁止此类中断发生。

串行终端的写函数 `rs_write()` 在 `serial.c` 程序中实现。串行中断程序在 `rs_io.s` 中实现。串行终端三个字符缓冲队列与函数、程序的关系参见图 7-5 所示。

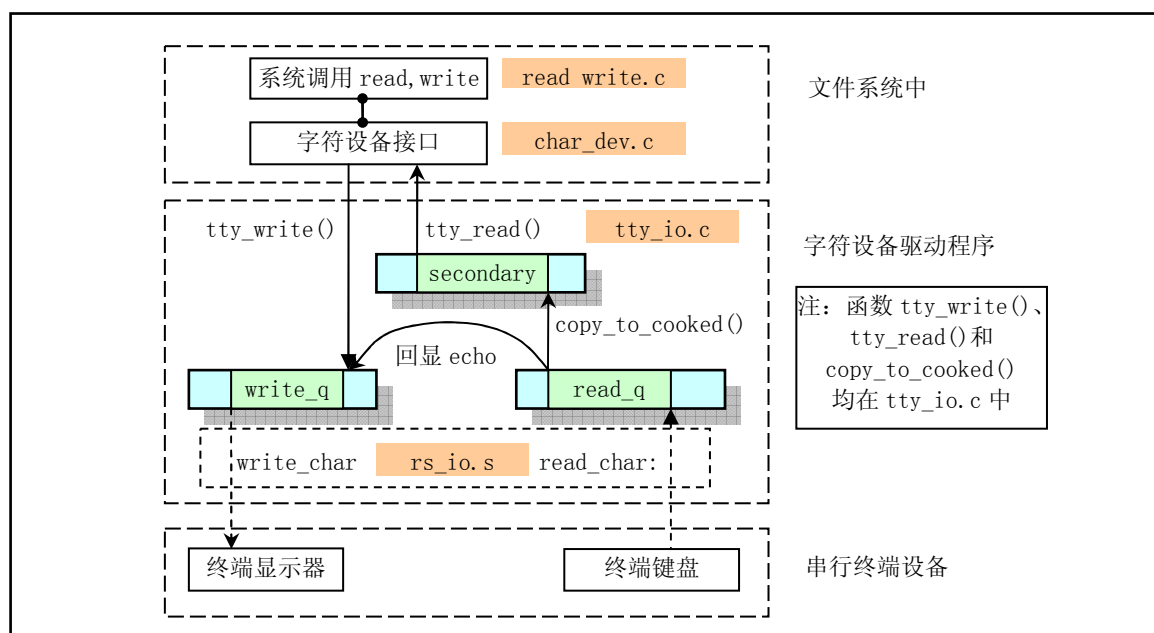


图 7-5 串行终端设备字符缓冲队列与函数之间的关系

由上图可见，串行终端与控制台处理过程之间的主要区别是串行终端利用程序 `rs_io.s` 取代了控制台操作显示器和键盘的程序 `console.c` 和 `keyboard.S`，其余部分的处理过程完全一样。

7.2.6 终端驱动程序接口

通常，用户是通过文件系统与设备打交道的，每个设备都有一个文件名称，相应地也在文件系统中占用一个索引节点 (i 节点)，但该 i 节点中的文件类型是设备类型，以便与其它正规文件相区别。用户就可以直接使用文件系统调用来访问设备。终端驱动程序也同样为此目的向文件系统提供了调用接口函数。终端驱动程序与系统其它程序的接口是使用 `tty_io.c` 文件中的通用函数实现的。其中实现了读终端函数 `tty_read()` 和写终端函数 `tty_write()`，以及输入行规则函数 `copy_to_cooked()`。另外，在 `tty_ioctl.c` 程序中，实现了修改终端参数的输入输出控制函数 (或系统调用) `tty_ioctl()`。终端的设置参数是放在终端数据结构中的 `termios` 结构中，其中的参数比较多，也比较复杂，请参考 `include/termios.h` 文件中的说明。

对于不同终端设备，可以有不同的行规则程序与之匹配。但在 Linux 0.11 中仅有一个行规则函数，因此 `termios` 结构中的行规则字段 `c_line` 不起作用，都被设置为 0。

7.3 Makefile 文件

7.3.1 功能描述

字符设备驱动程序的编译管理程序。由 Make 工具软件使用。

7.3.2 代码注释

程序 7-1 linux/kernel/chr_drv/Makefile

```

1 #
2 # Makefile for the FREAX-kernel character device drivers.
3 #
4 # Note! Dependencies are done automagically by 'make dep', which also
5 # removes any old dependencies. DON'T put your own dependencies here
6 # unless it's something special (ie not a .c file).
7 #
8 # FREAX(Linux) 内核字符设备驱动程序的 Makefile 文件。
9 # 注意！依赖关系是由 'make dep' 自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
10 # 依赖关系信息放在这里，除非是特别文件的（也即不是一个.c 文件的信息）。
11
12 AR      =gar      # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
13 AS      =gas      # GNU 的汇编程序。
14 LD      =gld      # GNU 的连接程序。
15 LDFLAGS =-s -x    # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
16 CC      =gcc      # GNU C 语言编译器。
17
18 # 下一行是 C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
19 # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
20 # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
21 # 单短小的函数代码嵌入调用程序中；-mstring-insns Linus 自己填加的优化选项，以后不再使用；
22 # -nostdinc -I../include 不使用默认路径中的包含文件，而使用指定目录中的(../../include)。
23 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
24         -finline-functions -mstring-insns -nostdinc -I../../include
25 # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
26 # 出设备或指定的输出文件中；-nostdinc -I../../include 同前。
27 CPP      =gcc -E -nostdinc -I../../include
28
29 # 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
30 # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
31 # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
32 # 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$.s（或$@）是自动目标变量，
33 # $<代表第一个先决条件，这里即是符合条件*.c 的文件。
34
35 .c.s:
36     $(CC) $(CFLAGS) \
37     -S -o $.s $<
38 # 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
39
40 .s.o:
41     $(AS) -c -o $.o $<
42
43 .c.o:
44     # 类似上面，*.c 文件→*.o 目标文件。不进行连接。
45     $(CC) $(CFLAGS) \
46     -c -o $.o $<
47

```

```

27 OBJS = tty_io.o console.o keyboard.o serial.o rs_io.o \    # 定义目标文件变量 OBJS。
28     tty_ioctl.o
29
30 chr_drv.a: $(OBJS)          # 在有了先决条件 OBJS 后使用下面的命令连接成目标 chr_drv.a 库文件。
31     $(AR) rcs chr_drv.a $(OBJS)
32     sync
33
34 # 对 keyboard.S 汇编程序进行预处理。-traditional 选项用来对程序作修改使其支持传统的 C 编译器。
35 # 处理后的程序改名为 kernboard.s。
36 keyboard.s: keyboard.S ../../include/linux/config.h
37     $(CPP) -traditional keyboard.S -o keyboard.s
38
39 # 下面的规则用于清理工作。当执行'make clean'时，就会执行下面的命令，去除所有编译
40 # 连接生成的文件。'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
41 clean:
42     rm -f core *.o *.a tmp_make keyboard.s
43     for i in *.c;do rm -f `basename $$i .c`.s;done
44
45 # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
46 # 使用字符串编辑程序 sed 对 Makefile 文件（即是本文件）进行处理，输出为删除 Makefile
47 # 文件中'### Dependencies'行后面的所有行（下面从 48 开始的行），并生成 tmp_make
48 # 临时文件（44 行的作用）。然后对 kernel/chr_drv/目录下的每个 C 文件执行 gcc 预处理操作。
49 # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
50 # 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
51 # 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
52 # 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
53 dep:
54     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
55     (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,\'`" "; \
56         $(CPP) -M $$i;done) >> tmp_make
57     cp tmp_make Makefile
58
59 ### Dependencies:
60 console.s console.o : console.c ../../include/linux/sched.h \
61     ../../include/linux/head.h ../../include/linux/fs.h \
62     ../../include/sys/types.h ../../include/linux/mm.h ../../include/signal.h \
63     ../../include/linux/tty.h ../../include/termios.h ../../include/asm/io.h \
64     ../../include/asm/system.h
65 serial.s serial.o : serial.c ../../include/linux/tty.h ../../include/termios.h \
66     ../../include/linux/sched.h ../../include/linux/head.h \
67     ../../include/linux/fs.h ../../include/sys/types.h ../../include/linux/mm.h \
68     ../../include/signal.h ../../include/asm/system.h ../../include/asm/io.h
69 tty_io.s tty_io.o : tty_io.c ../../include/ctype.h ../../include/errno.h \
70     ../../include/signal.h ../../include/sys/types.h \
71     ../../include/linux/sched.h ../../include/linux/head.h \
72     ../../include/linux/fs.h ../../include/linux/mm.h ../../include/linux/tty.h \
73     ../../include/termios.h ../../include/asm/segment.h \
74     ../../include/asm/system.h
75 tty_ioctl.s tty_ioctl.o : tty_ioctl.c ../../include/errno.h ../../include/termios.h \
76     ../../include/linux/sched.h ../../include/linux/head.h \
77     ../../include/linux/fs.h ../../include/sys/types.h ../../include/linux/mm.h \
78     ../../include/signal.h ../../include/linux/kernel.h \
79     ../../include/linux/tty.h ../../include/asm/io.h \

```

```
68 ../../include/asm/segment.h ../../include/asm/system.h
```

7.4 keyboard.s 程序

7.4.1 功能描述

该键盘驱动汇编程序主要包括键盘中断处理程序。在英文惯用法中，`make` 表示键被按下；`break` 表示键被松开(放开)。

该程序首先根据键盘特殊键（例如 `Alt`、`Shift`、`Ctrl`、`Caps` 键）的状态设置程序后面要用到的状态标志变量 `mode` 的值，然后根据引起键盘中断的按键扫描码，调用已经编排成跳转表的相应扫描码处理子程序，把扫描码对应的字符放入读字符队列(`read_q`)中。接下来调用 C 处理函数 `do_tty_interrupt()`(`tty_io.c`, 342 行)，该函数仅包含一个对行规程函数 `copy_to_cooked()` 的调用。这个行规程函数的主要作用就是把 `read_q` 读缓冲队列中的字符经过适当处理放入规范模式队列（辅助队列 `secondary`）中，并且在处理过程中，若相应终端设备设置了回显标志，还会把字符放入写队列（`write_q`）中，从而在终端屏幕上会显示出刚键入的字符。

对于 AT 键盘的扫描码，当键按下时，则对应键的扫描码被送出，但当键松开时，将会发送两个字节，第一个是 `0xf0`，第 2 个还是按下时的扫描码。为了向下的兼容性，设计人员将 AT 键盘发出的扫描码转换成了老式 PC/XT 标准键盘的扫描码。因此这里仅对 PC/XT 的扫描码进行处理即可。有关键盘扫描码的说明，请参见程序列表后的描述。

7.4.2 代码注释

程序 7-2 linux/kernel/chr_drv/keyboard.S

```

1 /*
2  * linux/kernel/keyboard.S
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * Thanks to Alfred Leung for US keyboard patches
9  * Wolfgang Thiel for German keyboard patches
10 * Marc Corsini for the French keyboard
11 */
12 /*
13 * 感谢 Alfred Leung 添加了 US 键盘补丁程序；
14 * Wolfgang Thiel 添加了德语键盘补丁程序；
15 * Marc Corsini 添加了法文键盘补丁程序。
16 */
17
18 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型 (HD_TYPE) 可选项。
19
20 .text
21 .globl _keyboard_interrupt
22

```

```

18 /*
19  * these are for the keyboard read functions
20  */
21 /*
22  * 以下这些是用于键盘读操作。
23  */
24 // size 是键盘缓冲区的长度（字节数）。
25 size = 1024 /* must be a power of two ! And MUST be the same
26 as in tty_io.c !!!! */
27 /* 数值必须是 2 的次方！并且与 tty_io.c 中的值匹配!!!! */
28 // 以下这些是缓冲队列结构中的偏移量 */
29 head = 4 // 缓冲区中头指针字段偏移。
30 tail = 8 // 缓冲区中尾指针字段偏移。
31 proc_list = 12 // 等待该缓冲队列的进程字段偏移。
32 buf = 16 // 缓冲区字段偏移。
33
34 // mode 是键盘特殊键的按下状态标志。
35 // 表示大小写转换键(caps)、交换键(alt)、控制键(ctrl)和换档键(shift)的状态。
36 // 位 7 caps 键按下；
37 // 位 6 caps 键的状态(应该与 leds 中的对应标志位一样)；
38 // 位 5 右 alt 键按下；
39 // 位 4 左 alt 键按下；
40 // 位 3 右 ctrl 键按下；
41 // 位 2 左 ctrl 键按下；
42 // 位 1 右 shift 键按下；
43 // 位 0 左 shift 键按下。
44 mode: .byte 0 /* caps, alt, ctrl and shift mode */
45 // 数字锁定键(num-lock)、大小写转换键(caps-lock)和滚动锁定键(scroll-lock)的 LED 发光管状态。
46 // 位 7-3 全 0 不用；
47 // 位 2 caps-lock；
48 // 位 1 num-lock(初始置 1，也即设置数字锁定键(num-lock)发光管为亮)；
49 // 位 0 scroll-lock。
50 leds: .byte 2 /* num-lock, caps, scroll-lock mode (nom-lock on) */
51 // 当扫描码是 0xe0 或 0xe1 时，置该标志。表示其后还跟随着 1 个或 2 个字符扫描码，参见列表后说明。
52 // 位 1 =1 收到 0xe1 标志；
53 // 位 0 =1 收到 0xe0 标志。
54 e0: .byte 0
55
56 /*
57  * con_int is the real interrupt routine that reads the
58  * keyboard scan-code and converts it into the appropriate
59  * ascii character(s).
60  */
61 /*
62  * con_int 是实际的中断处理子程序，用于读键盘扫描码并将其转换
63  * 成相应的 ascii 字符。
64  */
65 // 键盘中断处理程序入口点。
66 _keyboard_interrupt:
67     pushl %eax
68     pushl %ebx
69     pushl %ecx
70     pushl %edx

```

```

42     push %ds
43     push %es
44     movl $0x10,%eax      // 将 ds、es 段寄存器置为内核数据段。
45     mov %ax,%ds
46     mov %ax,%es
47     xorl %al,%al         /* %eax is scan code */ /* eax 中是扫描码 */
48     inb $0x60,%al        // 读取扫描码→al。
49     cmpb $0xe0,%al       // 该扫描码是 0xe0 吗？如果是则跳转到设置 e0 标志代码处。
50     je set_e0
51     cmpb $0xe1,%al       // 扫描码是 0xe1 吗？如果是则跳转到设置 e1 标志代码处。
52     je set_e1
53     call key_table(,%eax,4) // 调用键处理程序 ker_table + eax * 4 (参见下面 502 行)。
54     movb $0,e0           // 复位 e0 标志。
// 下面这段代码(55-65 行)是针对使用 8255A 的 PC 标准键盘电路进行硬件复位处理。端口 0x61 是
// 8255A 输出口 B 的地址，该输出端口的第 7 位 (PB7) 用于禁止和允许对键盘数据的处理。
// 这段程序用于对收到的扫描码做出应答。方法是首先禁止键盘，然后立刻重新允许键盘工作。
55 e0_e1:  inb $0x61,%al     // 取 PPI 端口 B 状态，其位 7 用于允许/禁止(0/1) 键盘。
56         jmp 1f           // 延迟一会。
57 1:      jmp 1f
58 1:      orb $0x80,%al     // al 位 7 置位(禁止键盘工作)。
59         jmp 1f           // 再延迟一会。
60 1:      jmp 1f
61 1:      outb %al,$0x61    // 使 PPI PB7 位置位。
62         jmp 1f           // 延迟一会。
63 1:      jmp 1f
64 1:      andb $0x7F,%al    // al 位 7 复位。
65         outb %al,$0x61    // 使 PPI PB7 位复位 (允许键盘工作)。
66         movb $0x20,%al    // 向 8259 中断芯片发送 EOI(中断结束)信号。
67         outb %al,$0x20
68         pushl $0          // 控制台 tty 号=0，作为参数入栈。
69         call _do_tty_interrupt // 将收到数据复制成规范模式数据并存放在规范字符缓冲队列中。
70         addl $4,%esp      // 丢弃入栈的参数，弹出保留的寄存器，并中断返回。
71         pop %es
72         pop %ds
73         popl %edx
74         popl %ecx
75         popl %ebx
76         popl %eax
77         iret
78 set_e0: movb $1,e0        // 收到扫描前导码 0xe0 时，设置 e0 标志 (位 0)。
79         jmp e0_e1
80 set_e1: movb $2,e0        // 收到扫描前导码 0xe1 时，设置 e1 标志 (位 1)。
81         jmp e0_e1
82
83 /*
84  * This routine fills the buffer with max 8 bytes, taken from
85  * %ebx:%eax. (%edx is high). The bytes are written in the
86  * order %al,%ah,%eal,%eah,%bl,%bh ... until %eax is zero.
87  */
/*
 * 下面该子程序把 ebx:eax 中的最多 8 个字符添入缓冲队列中。(edx 是
 * 所写入字符的顺序是 al, ah, eal, eah, bl, bh... 直到 eax 等于 0。
 */

```



```

88 put_queue:
89     pushl %ecx                // 保存 ecx, edx 内容。
90     pushl %edx                // 取控制台 tty 结构中读缓冲队列指针。
91     movl _table_list,%edx     # read-queue for console
92     movl head(%edx),%ecx      // 取缓冲队列中头指针→ecx。
93 1:   movb %al,buf(%edx,%ecx)  // 将 al 中的字符放入缓冲队列头指针位置处。
94     incl %ecx                // 头指针前移 1 字节。
95     andl $size-1,%ecx        // 以缓冲区大小调整头指针(若超出则返回缓冲区开始)。
96     cmpl tail(%edx),%ecx     # buffer full - discard everything
                                   // 头指针==尾指针吗(缓冲队列满)?
97     je 3f                    // 如果已满,则后面未放入的字符全抛弃。
98     shrdl $8,%ebx,%eax       // 将 ebx 中 8 位比特位右移 8 位到 eax 中,但 ebx 不变。
99     je 2f                    // 还有字符吗? 若没有(等于 0)则跳转。
100    shrll $8,%ebx            // 将 ebx 中比特位右移 8 位,并跳转到标号 1 继续操作。
101    jmp 1b
102 2:   movl %ecx,head(%edx)     // 若已将所有字符都放入了队列,则保存头指针。
103     movl proc_list(%edx),%ecx // 该队列的等待进程指针?
104     testl %ecx,%ecx          // 检测任务结构指针是否空(有等待该队列的进程吗?)。
105     je 3f                    // 无,则跳转;
106     movl $0, (%ecx)          // 有,则置该进程为可运行就绪状态(唤醒该进程)。
107 3:   popl %edx                // 弹出保留的寄存器并返回。
108     popl %ecx
109     ret
110
    // 下面这段代码根据 ctrl 或 alt 的扫描码,分别设置模式标志中相应位。如果该扫描码之前收到过
    // 0xe0 扫描码(e0 标志置位),则说明按下的是键盘右边的 ctrl 或 alt 键,则对应设置 ctrl 或 alt
    // 在模式标志 mode 中的比特位。
111 ctrl: movb $0x04,%al        // 0x4 是模式标志 mode 中左 ctrl 键对应的比特位(位 2)。
112     jmp 1f
113 alt:  movb $0x10,%al        // 0x10 是模式标志 mode 中左 alt 键对应的比特位(位 4)。
114 1:   cmpb $0,e0             // e0 标志置位了吗(按下的是右边的 ctrl 或 alt 键吗)?
115     je 2f                    // 不是则转。
116     addb %al,%al            // 是,则改成置相应右键的标志位(位 3 或位 5)。
117 2:   orb %al,mode           // 设置模式标志 mode 中对应的比特位。
118     ret
    // 这段代码处理 ctrl 或 alt 键松开的扫描码,对应复位模式标志 mode 中的比特位。在处理时要根据
    // e0 标志是否置位来判断是否是键盘右边的 ctrl 或 alt 键。
119 unctrl: movb $0x04,%al      // 模式标志 mode 中左 ctrl 键对应的比特位(位 2)。
120     jmp 1f
121 unalt: movb $0x10,%al      // 0x10 是模式标志 mode 中左 alt 键对应的比特位(位 4)。
122 1:   cmpb $0,e0             // e0 标志置位了吗(释放的是右边的 ctrl 或 alt 键吗)?
123     je 2f                    // 不是,则转。
124     addb %al,%al            // 是,则该成复位相应右键的标志位(位 3 或位 5)。
125 2:   notb %al               // 复位模式标志 mode 中对应的比特位。
126     andb %al,mode
127     ret
128
129 lshift:
130     orb $0x01,mode          // 是左 shift 键按下,设置 mode 中对应的标志位(位 0)。
131     ret
132 unlshift:
133     andb $0xfe,mode          // 是左 shift 键松开,复位 mode 中对应的标志位(位 0)。
134     ret

```

```

135 rshift:
136     orb $0x02,mode           // 是右 shift 键按下，设置 mode 中对应的标志位(位 1)。
137     ret
138 unrshift:
139     andb $0xfd,mode          // 是右 shift 键松开，复位 mode 中对应的标志位(位 1)。
140     ret
141
142 caps:   testb $0x80,mode      // 测试模式标志 mode 中位 7 是否已经置位(按下状态)。
143         jne 1f                // 如果已处于按下状态，则返回(ret)。
144         xorb $4,leds           // 翻转 leds 标志中 caps-lock 比特位(位 2)。
145         xorb $0x40,mode        // 翻转 mode 标志中 caps 键按下的比特位(位 6)。
146         orb $0x80,mode         // 设置 mode 标志中 caps 键已按下标志位(位 7)。
// 这段代码根据 leds 标志，开启或关闭 LED 指示器。
147 set_leds:
148     call kb_wait              // 等待键盘控制器输入缓冲空。
149     movb $0xed,%al            /* set leds command */ /* 设置 LED 的命令 */
150     outb %al,$0x60             // 发送键盘命令 0xed 到 0x60 端口。
151     call kb_wait              // 等待键盘控制器输入缓冲空。
152     movb leds,%al             // 取 leds 标志，作为参数。
153     outb %al,$0x60            // 发送该参数。
154     ret
155 uncaps: andb $0x7f,mode        // caps 键松开，则复位模式标志 mode 中的对应位(位 7)。
156     ret
157 scroll:
158     xorb $1,leds              // scroll 键按下，则翻转 leds 标志中的对应位(位 0)。
159     jmp set_leds              // 根据 leds 标志重新开启或关闭 LED 指示器。
160 num:   xorb $2,leds           // num 键按下，则翻转 leds 标志中的对应位(位 1)。
161     jmp set_leds              // 根据 leds 标志重新开启或关闭 LED 指示器。
162
163 /*
164  * curosr-key/numeric keypad cursor keys are handled here.
165  * checking for numeric keypad etc.
166  */
167 /*
168  * 这里处理方向键/数字小键盘方向键，检测数字小键盘等。
169  */
170 cursor:
171     subb $0x47,%al            // 扫描码是小数字键盘上的键(其扫描码>=0x47)发出的?
172     jb 1f                     // 如果小于则不处理，返回。
173     cmpb $12,%al              // 如果扫描码 > 0x53(0x53 - 0x47= 12)，则
174     ja 1f                     // 扫描码值超过 83(0x53)，不处理，返回。
175     jne cur2                  /* check for ctrl-alt-del */ /* 检查是否 ctrl-alt-del */
// 如果等于 12，则说明 del 键已被按下，则继续判断 ctrl
// 和 alt 是否也同时按下。
176     testb $0x0c,mode          // 有 ctrl 键按下吗?
177     je cur2                   // 无，则跳转。
178     testb $0x30,mode          // 有 alt 键按下吗?
179     jne reboot                // 有，则跳转到重启处理。
180 cur2:   cmpb $0x01,e0          /* e0 forces cursor movement */ /* e0 置位表示光标移动 */
// e0 标志置位了吗?
181     je cur                    // 置位了，则跳转光标移动处理处 cur。
182     testb $0x02,leds          /* not num-lock forces cursor */ /* num-lock 键则不许 */
// 测试 leds 中标志 num-lock 键标志是否置位。

```

```

180     je cur                // 如果没有置位(num 的 LED 不亮), 则也进行光标移动处理。
181     testb $0x03,mode      /* shift forces cursor */ /* shift 键也使光标移动 */
                                // 测试模式标志 mode 中 shift 按下标志。
182     jne cur              // 如果有 shift 键按下, 则也进行光标移动处理。
183     xorl %ebx,%ebx        // 否则查询扫数字表(199 行), 取对应键的数字 ASCII 码。
184     movb num_table(%eax),%al    // 以 eax 作为索引值, 取对应数字字符→al。
185     jmp put_queue        // 将该字符放入缓冲队列中。
186 1:     ret
187
    // 这段代码处理光标的移动。
188 cur:   movb cur_table(%eax),%al // 取光标字符表中相应键的代表字符→al。
189     cmpb $'9',%al          // 若该字符<='9', 说明是上一页、下一页、插入或删除键,
190     ja ok_cur              // 则功能字符序列中要添入字符'~'。
191     movb $'~',%ah
192 ok_cur: shll $16,%eax       // 将 ax 中内容移到 eax 高字中。
193     movw $0x5b1b,%ax       // 在 ax 中放入'esc [' 字符, 与 eax 高字中字符组成移动序列。
194     xorl %ebx,%ebx
195     jmp put_queue          // 将该字符放入缓冲队列中。
196
197 #if defined(KBD_FR)
198 num_table:
199     .ascii "789 456 1230."    // 数字小键盘上键对应的数字 ASCII 码表。
200 #else
201 num_table:
202     .ascii "789 456 1230,"
203 #endif
204 cur_table:
205     .ascii "HA5 DGC YB623"    // 数字小键盘上方向键或插入删除键对应的移动表示字符表。
206
207 /*
208  * this routine handles function keys
209  */
    // 下面子程序处理功能键。
210 func:
211     pushl %eax
212     pushl %ecx
213     pushl %edx
214     call _show_stat          // 调用显示各任务状态函数(kernl/sched.c, 37)。
215     popl %edx
216     popl %ecx
217     popl %eax
218     subb $0x3B,%al          // 功能键'F1'的扫描码是 0x3B, 因此此时 al 中是功能键索引号。
219     jb end_func             // 如果扫描码小于 0x3b, 则不处理, 返回。
220     cmpb $9,%al             // 功能键是 F1-F10?
221     jbe ok_func             // 是, 则跳转。
222     subb $18,%al            // 是功能键 F11, F12 吗?
223     cmpb $10,%al           // 是功能键 F11?
224     jb end_func             // 不是, 则不处理, 返回。
225     cmpb $11,%al           // 是功能键 F12?
226     ja end_func            // 不是, 则不处理, 返回。
227 ok_func:
228     cmpl $4,%ecx            /* check that there is enough room */ /* 检查是否有足够空间
*/

```

```

229         jl end_func                // 需要放入 4 个字符序列，如果放不下，则返回。
230         movl func_table(,%eax,4),%eax // 取功能键对应字符序列。
231         xorl %ebx,%ebx
232         jmp put_queue              // 放入缓冲队列中。
233 end_func:
234         ret
235
236 /*
237  * function keys send F1:'esc [ [ A' F2:'esc [ [ B' etc.
238  */
239 /*
240  * 功能键发送的扫描码，F1 键为：'esc [ [ A'， F2 键为：'esc [ [ B' 等。
241  */
242 func_table:
243         .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
244         .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
245         .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b
246
247 // 扫描码-ASCII 字符映射表。
248 // 根据在 config.h 中定义的键盘类型 (FINNISH, US, GERMEN, FRANCH)，将相应键的扫描码映射
249 // 到 ASCII 字符。
250 #if defined(KBD_FINNISH)
251 // 以下是芬兰语键盘的扫描码映射表。
252 key_map:
253         .byte 0,27                // 扫描码 0x00,0x01 对应的 ASCII 码;
254         .ascii "1234567890+' "    // 扫描码 0x02,...0x0c,0x0d 对应的 ASCII 码，以下类似。
255         .byte 127,9
256         .ascii "qwertyuiop}"
257         .byte 0,13,0
258         .ascii "asdfghjkl|{"
259         .byte 0,0
260         .ascii "'`zxcvbnm,.-"
261         .byte 0,'*',0,32          /* 36-39 */ /* 扫描码 0x36-0x39 对应的 ASCII 码 */
262         .fill 16,1,0              /* 3A-49 */ /* 扫描码 0x3A-0x49 对应的 ASCII 码 */
263         .byte '-',0,0,0,0,'+'    /* 4A-4E */ /* 扫描码 0x4A-0x4E 对应的 ASCII 码 */
264         .byte 0,0,0,0,0,0,0,0    /* 4F-55 */ /* 扫描码 0x4F-0x55 对应的 ASCII 码 */
265         .byte '<'
266         .fill 10,1,0
267
268 // shift 键同时按下时的映射表。
269 shift_map:
270         .byte 0,27
271         .ascii "!\"#$%&/()=?`"
272         .byte 127,9
273         .ascii "QWERTYUIOP]^"
274         .byte 13,0
275         .ascii "ASDFGHJKL\\["
276         .byte 0,0
277         .ascii "*ZXCVBNM;:_'"
278         .byte 0,'*',0,32          /* 36-39 */
279         .fill 16,1,0              /* 3A-49 */
280         .byte '-',0,0,0,0,'+'    /* 4A-4E */
281         .byte 0,0,0,0,0,0,0,0    /* 4F-55 */

```

```

274     .byte '>
275     .fill 10,1,0
276
    // alt 键同时按下时的映射表。
277 alt_map:
278     .byte 0,0
279     .ascii "\0@\0$\0\0{[]}\0"
280     .byte 0,0
281     .byte 0,0,0,0,0,0,0,0,0,0
282     .byte '~',13,0
283     .byte 0,0,0,0,0,0,0,0,0,0
284     .byte 0,0
285     .byte 0,0,0,0,0,0,0,0,0,0
286     .byte 0,0,0,0          /* 36-39 */
287     .fill 16,1,0          /* 3A-49 */
288     .byte 0,0,0,0,0        /* 4A-4E */
289     .byte 0,0,0,0,0,0,0    /* 4F-55 */
290     .byte '|
291     .fill 10,1,0
292
293 #elif defined(KBD_US)
294
    // 以下是美式键盘的扫描码映射表。
295 key_map:
296     .byte 0,27
297     .ascii "1234567890-="
298     .byte 127,9
299     .ascii "qwertyuiop[]"
300     .byte 13,0
301     .ascii "asdfghjkl;'"
302     .byte '`',0
303     .ascii "\\zxcvbnm,./"
304     .byte 0,'*',0,32      /* 36-39 */
305     .fill 16,1,0          /* 3A-49 */
306     .byte '-,0,0,0,0,0,0 /* 4A-4E */
307     .byte 0,0,0,0,0,0,0,0 /* 4F-55 */
308     .byte '<
309     .fill 10,1,0
310
311
312 shift_map:
313     .byte 0,27
314     .ascii "!@#$$%^&*()_+"
315     .byte 127,9
316     .ascii "QWERTYUIOP{}"
317     .byte 13,0
318     .ascii "ASDFGHJKL:"
319     .byte '~',0
320     .ascii "|ZXCVCBNM<>?"
321     .byte 0,'*',0,32      /* 36-39 */
322     .fill 16,1,0          /* 3A-49 */
323     .byte '-,0,0,0,0,0,0 /* 4A-4E */
324     .byte 0,0,0,0,0,0,0,0 /* 4F-55 */

```



```

325     .byte '>
326     .fill 10,1,0
327
328 alt_map:
329     .byte 0,0
330     .ascii "\0@\0$\0\0{[]}\0"
331     .byte 0,0
332     .byte 0,0,0,0,0,0,0,0,0,0
333     .byte '~',13,0
334     .byte 0,0,0,0,0,0,0,0,0,0
335     .byte 0,0
336     .byte 0,0,0,0,0,0,0,0,0,0
337     .byte 0,0,0,0          /* 36-39 */
338     .fill 16,1,0          /* 3A-49 */
339     .byte 0,0,0,0,0        /* 4A-4E */
340     .byte 0,0,0,0,0,0,0    /* 4F-55 */
341     .byte '|'
342     .fill 10,1,0
343
344 #elif defined(KBD_GR)
345
346 // 以下是德语键盘的扫描码映射表。
347 key_map:
348     .byte 0,27
349     .ascii "1234567890\\'"
350     .byte 127,9
351     .ascii "qwertzuiop@+"
352     .byte 13,0
353     .ascii "asdfghjkl[]^"
354     .byte 0,'#
355     .ascii "yxcvbnm,.-"
356     .byte 0,'*,0,32      /* 36-39 */
357     .fill 16,1,0          /* 3A-49 */
358     .byte '-,0,0,0,'+    /* 4A-4E */
359     .byte 0,0,0,0,0,0,0  /* 4F-55 */
360     .byte '<'
361     .fill 10,1,0
362
363 shift_map:
364     .byte 0,27
365     .ascii "!\"#$%&/()=?`"
366     .byte 127,9
367     .ascii "QWERTZUIOP\\*"
368     .byte 13,0
369     .ascii "ASDFGHJKL{}~"
370     .byte 0,''
371     .ascii "YXCVBNM;:_-"
372     .byte 0,'*,0,32      /* 36-39 */
373     .fill 16,1,0          /* 3A-49 */
374     .byte '-,0,0,0,'+    /* 4A-4E */
375     .byte 0,0,0,0,0,0,0  /* 4F-55 */
376     .byte '>'

```

```

377     .fill 10,1,0
378
379 alt_map:
380     .byte 0,0
381     .ascii "\0@\0$\0\0{[]}\0"
382     .byte 0,0
383     .byte '@,0,0,0,0,0,0,0,0,0,0
384     .byte '~ ,13,0
385     .byte 0,0,0,0,0,0,0,0,0,0,0
386     .byte 0,0
387     .byte 0,0,0,0,0,0,0,0,0,0,0
388     .byte 0,0,0,0          /* 36-39 */
389     .fill 16,1,0          /* 3A-49 */
390     .byte 0,0,0,0,0        /* 4A-4E */
391     .byte 0,0,0,0,0,0,0    /* 4F-55 */
392     .byte '|'
393     .fill 10,1,0
394
395
396 #elif defined(KBD_FR)
397
398 // 以下是法语键盘的扫描码映射表。
398 key_map:
399     .byte 0,27
400     .ascii "&{\"} (-)_/@="
401     .byte 127,9
402     .ascii "azertyuiop`$"
403     .byte 13,0
404     .ascii "qsd fghjklm|"
405     .byte ` ,0,42          /* coin sup gauche, don't know, [*|mu] */
406     .ascii "wxcvbn,;:!"
407     .byte 0,'*',0,32        /* 36-39 */
408     .fill 16,1,0          /* 3A-49 */
409     .byte '-,0,0,0,0,'+     /* 4A-4E */
410     .byte 0,0,0,0,0,0,0,0  /* 4F-55 */
411     .byte '<'
412     .fill 10,1,0
413
414 shift_map:
415     .byte 0,27
416     .ascii "1234567890]+'"
417     .byte 127,9
418     .ascii "AZERTYUIOP<>"
419     .byte 13,0
420     .ascii "QSDFGHJKLM%"
421     .byte '~ ,0,'#
422     .ascii "WXCVCBN?./\\"
423     .byte 0,'*',0,32        /* 36-39 */
424     .fill 16,1,0          /* 3A-49 */
425     .byte '-,0,0,0,0,'+     /* 4A-4E */
426     .byte 0,0,0,0,0,0,0,0  /* 4F-55 */
427     .byte '>'
428     .fill 10,1,0

```

```

429
430 alt_map:
431     .byte 0,0
432     .ascii "\0~#[[|`\\"@]}"
433     .byte 0,0
434     .byte '@,0,0,0,0,0,0,0,0,0,0
435     .byte '~',13,0
436     .byte 0,0,0,0,0,0,0,0,0,0,0
437     .byte 0,0
438     .byte 0,0,0,0,0,0,0,0,0,0,0
439     .byte 0,0,0,0          /* 36-39 */
440     .fill 16,1,0          /* 3A-49 */
441     .byte 0,0,0,0,0        /* 4A-4E */
442     .byte 0,0,0,0,0,0,0    /* 4F-55 */
443     .byte '|'
444     .fill 10,1,0
445
446 #else
447 #error "KBD-type not defined"
448 #endif
449 /*
450  * do_self handles "normal" keys, ie keys that don't change meaning
451  * and which have just one character returns.
452  */
453 /*
454  * do_self 用于处理“普通”键，也即含义没有变化并且只有一个字符返回的键。
455  */
456 do_self:
457     // 454-460 行用于根据模式标志 mode 选择 alt_map、shift_map 或 key_map 映射表之一。
458     lea alt_map,%ebx          // alt 键同时按下时的映射表基址 alt_map→ebx。
459     testb $0x20,mode         /* alt-gr */ /* 右 alt 键同时按下了? */
460     jne 1f                  // 是，则向前跳转到标号 1 处。
461     lea shift_map,%ebx       // shift 键同时按下时的映射表基址 shift_map→ebx。
462     testb $0x03,mode         // 有 shift 键同时按下了吗?
463     jne 1f                  // 有，则向前跳转到标号 1 处。
464     lea key_map,%ebx         // 否则使用普通映射表 key_map。
465     // 取映射表中对应扫描码的 ASCII 字符，若没有对应字符，则返回(转 none)。
466 1:     movb (%ebx,%eax),%al    // 将扫描码作为索引值，取对应的 ASCII 码→al。
467     orb %al,%al              // 检测看是否有对应的 ASCII 码。
468     je none                  // 若没有(对应的 ASCII 码=0)，则返回。
469     // 若 ctrl 键已按下或 caps 键锁定，并且字符在'a'-'}' (0x61-0x7D) 范围内，则将其转成大写字符
470     // (0x41-0x5D)。
471     testb $0x4c,mode         /* ctrl or caps */ /* 控制键已按下或 caps 亮? */
472     je 2f                  // 没有，则向前跳转标号 2 处。
473     cmpb $'a',%al            // 将 al 中的字符与'a'比较。
474     jb 2f                  // 若 al 值<'a'，则转标号 2 处。
475     cmpb $'}',%al            // 将 al 中的字符与'}'比较。
476     ja 2f                  // 若 al 值>'}',则转标号 2 处。
477     subb $32,%al             // 将 al 转换为大写字符(减 0x20)。
478     // 若 ctrl 键已按下，并且字符在'`'-'_' (0x40-0x5F)之间(是大写字符)，则将其转换为控制字符
479     // (0x00-0x1F)。
480 2:     testb $0x0c,mode         /* ctrl */ /* ctrl 键同时按下了吗? */
481     je 3f                  // 若没有则转标号 3。

```

```

473      cmpb $64,%al          // 将 al 与 '@' (64) 字符比较(即判断字符所属范围)。
474      jb 3f                 // 若值<'@'，则转标号 3。
475      cmpb $64+32,%al       // 将 al 与 '`' (96) 字符比较(即判断字符所属范围)。
476      jae 3f                // 若值>='`'，则转标号 3。
477      subb $64,%al          // 否则 al 值减 0x40，
                              // 即将字符转换为 0x00-0x1f 之间的控制字符。

// 若左 alt 键同时按下，则将字符的位 7 置位。
478 3:      testb $0x10,mode    /* left alt */ /* 左 alt 键同时按下? */
479      je 4f                 // 没有，则转标号 4。
480      orb $0x80,%al         // 字符的位 7 置位。
// 将 al 中的字符放入读缓冲队列中。
481 4:      andl $0xff,%eax      // 清 eax 的高字和 ah。
482      xorl %ebx,%ebx         // 清 ebx。
483      call put_queue         // 将字符放入缓冲队列中。
484 none:   ret
485
486 /*
487 * minus has a routine of it's own, as a 'E0h' before
488 * the scan code for minus means that the numeric keypad
489 * slash was pushed.
490 */
/*
* 减号有它自己的处理子程序，因为在减号扫描码之前的 0xe0
* 意味着按下了数字小键盘上的斜杠键。
*/
491 minus:  cmpb $1,e0          // e0 标志置位了吗？
492      jne do_self            // 没有，则调用 do_self 对减号符进行普通处理。
493      movl $'/',%eax         // 否则用 '/' 替换减号 '-' → al。
494      xorl %ebx,%ebx
495      jmp put_queue          // 并将字符放入缓冲队列中。
496
497 /*
498 * This table decides which routine to call when a scan-code has been
499 * gotten. Most routines just call do_self, or none, depending if
500 * they are make or break.
501 */
/* 下面是一张子程序地址跳转表。当取得扫描码后就根据此表调用相应的扫描码处理子程序。
* 大多数调用的子程序是 do_self，或者是 none，这取决于按键(make)还是释放键(break)。
*/
502 key_table:
503      .long none,do_self,do_self,do_self    /* 00-03 s0 esc 1 2 */
504      .long do_self,do_self,do_self,do_self /* 04-07 3 4 5 6 */
505      .long do_self,do_self,do_self,do_self /* 08-0B 7 8 9 0 */
506      .long do_self,do_self,do_self,do_self /* 0C-0F + ' bs tab */
507      .long do_self,do_self,do_self,do_self /* 10-13 q w e r */
508      .long do_self,do_self,do_self,do_self /* 14-17 t y u i */
509      .long do_self,do_self,do_self,do_self /* 18-1B o p } ^ */
510      .long do_self,ctrl,do_self,do_self    /* 1C-1F enter ctrl a s */
511      .long do_self,do_self,do_self,do_self /* 20-23 d f g h */
512      .long do_self,do_self,do_self,do_self /* 24-27 j k l | */
513      .long do_self,do_self,lshift,do_self  /* 28-2B { para lshift , */
514      .long do_self,do_self,do_self,do_self /* 2C-2F z x c v */
515      .long do_self,do_self,do_self,do_self /* 30-33 b n m , */

```

```

516 . long do_self, minus, rshift, do_self /* 34-37 . - rshift */
517 . long alt, do_self, caps, func /* 38-3B alt sp caps f1 */
518 . long func, func, func, func /* 3C-3F f2 f3 f4 f5 */
519 . long func, func, func, func /* 40-43 f6 f7 f8 f9 */
520 . long func, num, scroll, cursor /* 44-47 f10 num scr home */
521 . long cursor, cursor, do_self, cursor /* 48-4B up pgup - left */
522 . long cursor, cursor, do_self, cursor /* 4C-4F n5 right + end */
523 . long cursor, cursor, cursor, cursor /* 50-53 dn pgdn ins del */
524 . long none, none, do_self, func /* 54-57 sysreq ? < f11 */
525 . long func, none, none, none /* 58-5B f12 ? ? ? */
526 . long none, none, none, none /* 5C-5F ? ? ? ? */
527 . long none, none, none, none /* 60-63 ? ? ? ? */
528 . long none, none, none, none /* 64-67 ? ? ? ? */
529 . long none, none, none, none /* 68-6B ? ? ? ? */
530 . long none, none, none, none /* 6C-6F ? ? ? ? */
531 . long none, none, none, none /* 70-73 ? ? ? ? */
532 . long none, none, none, none /* 74-77 ? ? ? ? */
533 . long none, none, none, none /* 78-7B ? ? ? ? */
534 . long none, none, none, none /* 7C-7F ? ? ? ? */
535 . long none, none, none, none /* 80-83 ? br br br */
536 . long none, none, none, none /* 84-87 br br br br */
537 . long none, none, none, none /* 88-8B br br br br */
538 . long none, none, none, none /* 8C-8F br br br br */
539 . long none, none, none, none /* 90-93 br br br br */
540 . long none, none, none, none /* 94-97 br br br br */
541 . long none, none, none, none /* 98-9B br br br br */
542 . long none, unctrl, none, none /* 9C-9F br unctrl br br */
543 . long none, none, none, none /* A0-A3 br br br br */
544 . long none, none, none, none /* A4-A7 br br br br */
545 . long none, none, unlshift, none /* A8-AB br br unlshift br */
546 . long none, none, none, none /* AC-AF br br br br */
547 . long none, none, none, none /* B0-B3 br br br br */
548 . long none, none, unrshift, none /* B4-B7 br br unrshift br */
549 . long unalt, none, uncaps, none /* B8-BB unalt br uncaps br */
550 . long none, none, none, none /* BC-BF br br br br */
551 . long none, none, none, none /* C0-C3 br br br br */
552 . long none, none, none, none /* C4-C7 br br br br */
553 . long none, none, none, none /* C8-CB br br br br */
554 . long none, none, none, none /* CC-CF br br br br */
555 . long none, none, none, none /* D0-D3 br br br br */
556 . long none, none, none, none /* D4-D7 br br br br */
557 . long none, none, none, none /* D8-DB br ? ? ? */
558 . long none, none, none, none /* DC-DF ? ? ? ? */
559 . long none, none, none, none /* E0-E3 e0 e1 ? ? */
560 . long none, none, none, none /* E4-E7 ? ? ? ? */
561 . long none, none, none, none /* E8-EB ? ? ? ? */
562 . long none, none, none, none /* EC-EF ? ? ? ? */
563 . long none, none, none, none /* F0-F3 ? ? ? ? */
564 . long none, none, none, none /* F4-F7 ? ? ? ? */
565 . long none, none, none, none /* F8-FB ? ? ? ? */
566 . long none, none, none, none /* FC-FF ? ? ? ? */
567 /*
568 */

```



```

569 * kb_wait waits for the keyboard controller buffer to empty.
570 * there is no timeout - if the buffer doesn't empty, we hang.
571 */
/*
* 子程序 kb_wait 用于等待键盘控制器缓冲空。不存在超时处理 - 如果
* 缓冲永远不空的话，程序就会永远等待(死掉)。
*/
572 kb_wait:
573     pushl %eax
574 1:     inb $0x64,%al           // 读键盘控制器状态。
575     testb $0x02,%al         // 测试输入缓冲器是否为空(等于 0)。
576     jne 1b                  // 若不空，则跳转循环等待。
577     popl %eax
578     ret
579 /*
580 * This routine reboots the machine by asking the keyboard
581 * controller to pulse the reset-line low.
582 */
/*
* 该子程序通过设置键盘控制器，向复位线输出负脉冲，使系统复位重启(reboot)。
*/
583 reboot:
584     call kb_wait             // 首先等待键盘控制器输入缓冲器空。
585     movw $0x1234,0x472       /* don't do memory check */
586     movb $0xfc,%al          /* pulse reset and A20 low */
587     outb %al,$0x64           // 向系统复位和 A20 线输出负脉冲。
588 die:    jmp die              // 死机。

```

7.4.3 其它信息

7.4.3.1 AT 键盘接口编程

主机系统板上所采用的键盘控制器是 intel 8042 芯片或其兼容芯片，其逻辑示意图，见图 7-6 所示。其中输出端口 P2 分别用于其它目的。位 0(P20 引脚)用于实现 CPU 的复位操作，位 1 (P21 引脚)用于控制 A20 信号线的开启与否。当该输出端口位 0 为 1 时就开启(选通)了 A20 信号线，为 0 则禁止 A20 信号线。参见引导启动程序一章中对 A20 信号线的详细说明。

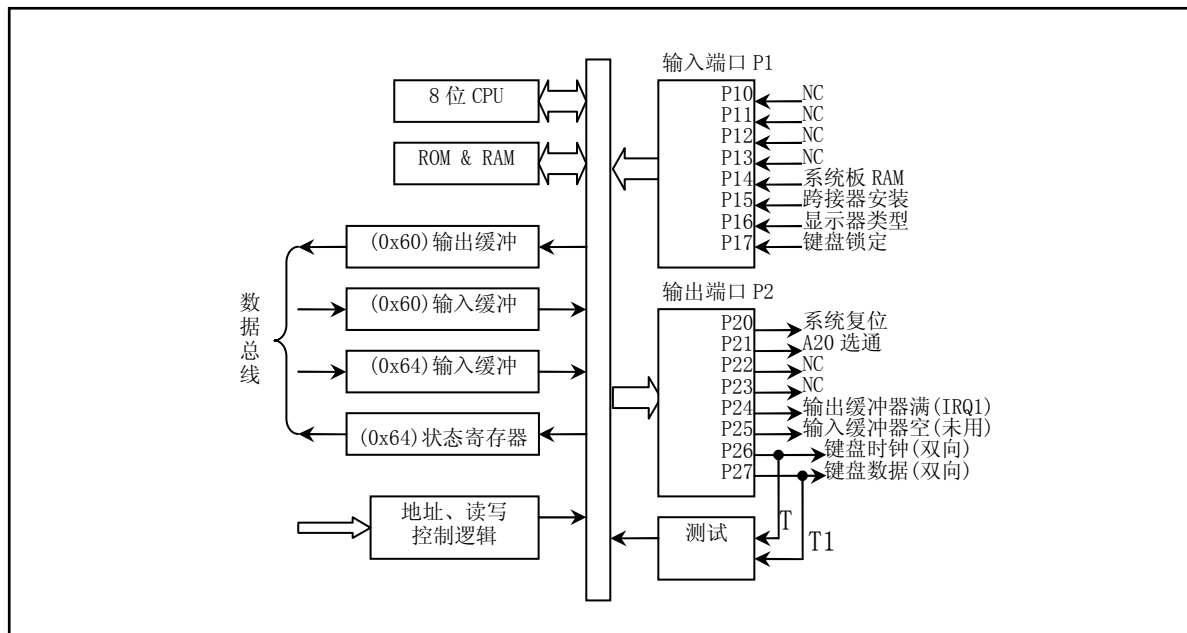


图 7-6 键盘控制器 804X 逻辑示意图

分配给键盘控制器的 IO 端口范围是 0x60-0x6f，但实际上 IBM CP/AT 使用的只有 0x60 和 0x64 两个口地址(0x61、0x62 和 0x63 用于与 XT 兼容目的)见表 7-1 所示，加上对端口的读和写操作含义不同，因此主要可有 4 种不同操作。对键盘控制器进行编程，将涉及芯片中的状态寄存器、输入缓冲器和输出缓冲器。

表 7-1 键盘控制器 804X 端口

端口	读/写	名称	用途
0x60	读	数据端口或输出缓冲器	是一个 8 位只读寄存器。当键盘控制器收到来自键盘的扫描码或命令响应时，一方面置状态寄存器位 0 = 1，另一方面产生中断 IRQ1。通常应该仅在状态端口位 0 = 1 时才读。
0x60	写	输入缓冲器	用于向键盘发送命令与/或随后的参数，或向键盘控制器写参数。键盘命令共有 10 多条，见表格后说明。通常都应该仅在状态端口位 1=0 时才写。
0x61	读/写		该端口 0x61 是 8255A 输出口 B 的地址，是针对使用/兼容 8255A 的 PC 标准键盘电路进行硬件复位处理。该端口用于对收到的扫描码做出应答。方法是首先禁止键盘，然后立刻重新允许键盘。所操作的数据为： 位 7=1 禁止键盘；=0 允许键盘； 位 6=0 迫使键盘时钟为低位，因此键盘不能发送任何数据。 位 5-0 这些位与键盘无关，是用于可编程并行接口(PPI)。
0x64	读	状态寄存器	是一个 8 位只读寄存器，其位字段含义分别为： 位 7=1 来自键盘传输数据奇偶校验错； 位 6=1 接收超时(键盘传送未产生 IRQ1)； 位 5=1 发送超时(键盘无响应)； 位 4=1 键盘接口被键盘锁禁止；[??是=0 时] 位 3=1 写入输入缓冲器中的数据是命令(通过端口 0x64)； =0 写入输入缓冲器中的数据是参数(通过端口 0x60)； 位 2 系统标志状态：0 = 上电启动或复位；1 = 自检通过；

			位 1=1 输入缓冲器满(0x60/64 口有给 8042 的数据); 位 0=1 输出缓冲器满(数据端口 0x60 有给系统的数据)。
0x64	写	输入缓冲器	向键盘控制器写命令。可带一参数, 参数从端口 0x60 写入。键盘控制器命令有 12 条, 见表格后说明。

7.4.3.2 键盘命令

系统在向端口 0x60 写入 1 字节, 便是发送键盘命令。键盘在接收到命令后 20ms 内应予以响应, 即回送一个命令响应。有的命令后还需要跟一参数 (也写到该端口)。命令列表见表 7-2 所示。注意, 如果没有另外指明, 所有命令均被回送一个 0xfa 响应码(ACK)。

表 7-2 键盘命令一览表

命令码	参数	功能
0xed	有	设置/复位模式指示器。置 1 开启, 0 关闭。参数字节: 位 7-3 保留全为 0; 位 2 = caps-lock 键; 位 1 = num-lock 键; 位 0 = scroll-lock 键。
0xee	无	诊断回应。键盘应回送 0xee。
0xef		保留不用。
0xf0	有	读取/设置扫描码集。参数字节等于: 0x00 - 选择当前扫描码集; 0x01 - 选择扫描码集 1(用于 PCs, PS/2 30 等); 0x02 - 选择扫描码集 2(用于 AT, PS/2, 是缺省值); 0x03 - 选择扫描码集 3。
0xf1		保留不用。
0xf2	无	读取键盘标识号(读取 2 个字节)。AT 键盘返回响应码 0xfa。
0xf3	有	设置扫描码连续发送时的速率和延迟时间。参数字节的含义为: 位 7 保留为 0; 位 6-5 延时值: 令 C=位 6-5, 则有公式: 延时值=(1+C)*250ms; 位 4-0 扫描码连续发送的速率; 令 B=位 4-3; A=位 2-0, 则有公式: 速率=1/((8+A)*2^B*0.00417)。 参数缺省值为 0x2c。
0xf4	无	开启键盘。
0xf5	无	禁止键盘。
0xf6	无	设置键盘默认参数。
0xf7-0xfd		保留不用。
0xfe	无	重发扫描码。当系统检测到键盘传输数据有错, 则发此命令。
0xff	无	执行键盘上电复位操作, 称之为基本保证测试(BAT)。操作过程为: 1. 键盘收到该命令后立刻响应发送 0xfa; 2. 键盘控制器使键盘时钟和数据线置为高电平; 3. 键盘开始执行 BAT 操作; 4. 若正常完成, 则键盘发送 0xaa; 否则发送 0xfd 并停止扫描。

7.4.3.3 键盘控制器命令

系统向输入缓冲(端口 0x64)写入 1 字节,即发送一键盘控制器命令。可带一参数。参数是通过写 0x60 端口发送的。见表 7-3 所示。

表 7-3 键盘控制器命令一览表

命令	参数	功能
0x20	无	读给键盘控制器的最后一个命令字节,放在端口 0x60 供系统读取。
0x21-0x3f	无	读取由命令低 5 比特位指定的控制器内部 RAM 中的命令。
0x60-0x7f	有	写键盘控制器命令字节。参数字节:(默认值为 0x5d) 位 7 保留为 0; 位 6 IBM PC 兼容模式(奇偶检验,转换为系统扫描码,单字节 PC 断开码); 位 5 PC 模式(对扫描码不进行奇偶校验;不转换成系统扫描码); 位 4 禁止键盘工作(使键盘时钟为低电平); 位 3 禁止超越(override),对键盘锁定转换不起作用; 位 2 系统标志;1 表示控制器工作正确; 位 1 保留为 0; 位 0 允许输出寄存器满中断。
0xaa	无	初始化键盘控制器自测试。成功返回 0x55;失败返回 0xfc。
0xab	无	初始化键盘接口测试。返回字节: 0x00 无错; 0x01 键盘时钟线为低(始终为低,低粘连); 0x02 键盘时钟线为高; 0x03 键盘数据线为低; 0x04 键盘数据线为高;
0xac	无	诊断转储。804x 的 16 字节 RAM、输出口、输入口状态依次输出给系统。
0xad	无	禁止键盘工作(设置命令字节位 4=1)。
0xae	无	允许键盘工作(复位命令字节位 4=0)。
0xc0	无	读 804x 的输入端口 P1,并放在 0x60 供读取;
0xd0	无	读 804x 的输出端口 P2,并放在 0x60 供读取;
0xd1	有	写 804x 的输出端口 P2,原 IBM PC 使用输出端口的位 2 控制 A20 门。注意,位 0(系统复位)应该总是置位的。
0xe0	无	读测试端 T0 和 T1 的输入送输出缓冲器供系统读取。 位 1 键盘数据;位 0 键盘时钟。
0xed	有	控制 LED 的状态。置 1 开启,0 关闭。参数字节: 位 7-3 保留全为 0; 位 2 = caps-lock 键; 位 1 = num-lock 键; 位 0 = scroll-lock 键。
0xf0-0xff	无	送脉冲到输出端口。该命令序列控制输出端口 P20-23 线,参见键盘控制器逻辑示意图。欲让哪一位输出负脉冲(6 微秒),即置该位为 0。也即该命令的低 4 位分别控制负脉冲的输出。例如,若要复位系统,则需发出命令 0xfe(P20 低)即可。

7.4.3.4 键盘扫描码

PC 机采用的均是非编码键盘。键盘上每个键都有一个位置编号,是从左到右从上到下。并且 PC XT

机与 AT 机键盘的位置码差别很大。键盘内的微处理机向系统发送的是键对应的扫描码。当键按下时，键盘输出的扫描码称为接通(make)扫描码，而该键松开时发送的则称为断开(break)扫描码。XT 键盘各键的扫描码见表 7-4 所示。

键盘上的每个键都有一个包含在字节低 7 位（位 6-0）中相应的扫描码。在高位（位 7）表示是按键还是松开按键。位 7=0 表示刚将键按下的扫描码，位 7=1 表示键松开的扫描码。例如，如果某人刚把 ESC 键按下，则传输给系统的扫描码将是 1（1 是 ESC 键的扫描码），当该键释放时将产生 $1+0x80=129$ 扫描码。

对于 PC、PC/XT 的标准 83 键键盘，接通扫描码与键号（键的位置码）是一样的。并用 1 字节表示。例如“A”键，键位置号是 30，接通码是扫描码是 0x1e。而其断开码是是接通扫描码加上 0x80，即 0x9e。对于 AT 机使用的 84/101/102 扩展键盘，则与 PC/XT 标准键盘区别较大。

对于某些“扩展”的键，则情况有些不同。当一个扩展键被按下时，将产生一个中断并且键盘端口将输出一个“扩展的”的扫描码前缀 0xe0，而在下一个“中断”中将给出。比如，对于 PC/XT 标准键盘，左边的控制键 ctrl 的扫描码是 29，而右边的“扩展的”控制键 ctrl 则具有一个扩展的扫描码 29。这个规则同样适合于 alt、箭头键。

另外，还有两个键的处理是非常特殊的。PrtScn 键和 Pause/Break 键。按下 PrtScn 键将会向键盘中断程序发送*2*个扩展字符，42(0x2a)和 55(0x37)，所以实际的字节序列将是 0xe0, 0x2a, 0xe0, 0x37。但在键重复产生时将只发送扩展码 0x37。当键松开时，又重新发送两个扩展的加上 0x80 的码（0xe0, 0xaa, 0xe0, 0xb7）。当 prtscn 键按下时，如果 shift 或 ctrl 键也按下了，则仅发送 0xe0, 0x37，并且在松开时仅发送 0xe0, 0xb7)。

对于 Pause/Break 键。如果你在按下该键的同时也按下了控制键，则将行如扩展键 70，而在其它情况下它将发送字符序列 0xe1, 0x1d, 0x45, 0xe1, 0x9d, 0xc5。将键按下并不会产生重复的扫描码，而松开键也并不会产生任何扫描码。

因此，可以这样来看待和处理：扫描码 0xe0 意味着还有一个字符跟随其后，而扫描码 0xe1 则表示后面跟着 2 个字符。

对于 AT 键盘的扫描码，与 PC/XT 的略有不同。当键按下时，则对应键的扫描码被送出，但当键松开时，将会发送两个字节，第一个是 0xf0，第 2 个还是相同的键扫描码。现在键盘设计者使用 8049 作为 AT 键盘的输入处理器，为了向下的兼容性将 AT 键盘发出的扫描码转换成了老式 PC/XT 标准键盘的扫描码。

AT 键盘有三种独立的扫描码集：一种是我们上面说明的(83 键映射，而增加的键有多余的 0xe0 码)，一种几乎是顺序的，还有一种却只有 1 个字节！最后一种所带来的问题是只有左 shift, caps, 左 ctrl 和左 alt 键的松开码被发送。键盘的默认扫描码集是扫描码集 2，可以利用命令更改。

对于扫描码集 1 和 2，有特殊码 0xe0 和 0xe1。它们用于具有相同功能的键。比如：左控制键 ctrl 位置是 0x1d(对于 PC/XT)，则右边的控制键就是 0xe0, 0x1d。这是为了与 PC/XT 程序兼容。请注意唯一使用 0xe1 的时候是当它表示临时控制键时，对此情况同时也有一个 0xe0 的版本。

表 7-4 XT 键盘扫描码表

F1	F2	`	1	2	3	4	5	6	7	8	9	0	-	=	\	BS	ESC	NUML	SCRL	SYSR
3B	3C	29	02	03	04	05	06	07	08	09	0A	0B	0C	0D	2B	0E	01	45	46	**
F3	F4	TAB	Q	W	E	R	T	Y	U	I	O	P	[]			Home	↑	PgUp	PrtSc
3D	3E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B			47	48	49	37
F5	F6	CNTL	A	S	D	F	G	H	J	K	L	;	'		ENTER		←	5	→	-
3F	40	1D	1E	1F	20	21	22	23	24	25	26	27	28	1C			4B	4C	4D	4A
F7	F8	LSHFT	Z	X	C	V	B	N	M	,	.	/		RSHFT			End	↓	PgDn	+
41	42	2A		2C	2D	2E	2F	30	31	32	33	34	35	36			4F	50	51	4E
F9	F0	ALT		Space												CAPLOCK	Ins		Del	
3F	40	1D		39												3A	52		53	

7.5 console.c 程序

7.5.1 功能描述

本文件是内核中最长的程序之一，但功能比较单一。其中的所有子程序都是为了实现终端屏幕写函数 `con_write()` 以及进行终端初始化操作。

函数 `con_write()` 会从终端 `tty_struct` 结构的写缓冲队列 `write_q` 中取出字符或字符序列，然后根据字符的性质（是普通字符还是转义字符序列），把字符显示在终端屏幕上或进行一些光标移动、字符擦除等屏幕控制操作。

终端屏幕初始化函数 `con_init()` 会根据系统初始化时获得的系统信息，设置有关屏幕的一些基本参数值，用于 `con_write()` 函数的操作。

有关终端设备字符缓冲队列的说明可参见 `include/linux/tty.h` 头文件。其中给出了字符缓冲队列的数据结构 `tty_queue`、终端的数据结构 `tty_struct` 和一些控制字符的值。另外还有一些对缓冲队列进行操作的宏定义。缓冲队列及其操作示意图请参见图 7-9 所示。

7.5.2 代码注释

程序 7-3 linux/kernel/chr_drv/console.c

```

1  /*
2   *  linux/kernel/console.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *      console.c
9   *
10  *  This module implements the console io functions
11  *      'void con_init(void)'
12  *      'void con_write(struct tty_queue * queue)'
13  *  Hopefully this will be a rather complete VT102 implementation.
14  *
15  *  Beeping thanks to John T Kohl.
16  */
17  /*
18  * 该模块实现控制台输入输出功能
19  *      'void con_init(void)'
20  *      'void con_write(struct tty_queue * queue)'
21  * 希望这是一个非常完整的 VT102 实现。
22  *
23  * 感谢 John T Kohl 实现了蜂鸣指示。
24  */
25
26  /*
27  *  NOTE!!! We sometimes disable and enable interrupts for a short while
28  *  (to put a word in video IO), but this will work even for keyboard

```

```

21  * interrupts. We know interrupts aren't enabled when getting a keyboard
22  * interrupt, as we use trap-gates. Hopefully all is well.
23  */
/*
  * 注意!!! 我们有时短暂地禁止和允许中断(在将一个字(word)放到视频 IO), 但即使
  * 对于键盘中断这也是可以工作的。因为我们使用陷阱门, 所以我们知道在获得一个
  * 键盘中断时中断是不允许的。希望一切均正常。
  */
24
25 /*
26  * Code to check for different video-cards mostly by Galen Hunt,
27  * <g-hunt@ee.utah.edu>
28  */
/*
  * 检测不同显示卡的代码大多数是 Galen Hunt 编写的,
  * <g-hunt@ee.utah.edu>
  */
29
30 #include <linux/sched.h> // 调度程序头文件, 定义了任务结构 task_struct、初始任务 0 的数据,
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
31 #include <linux/tty.h> // tty 头文件, 定义了有关 tty_io, 串行通信方面的参数、常数。
32 #include <asm/io.h>    // io 头文件。定义硬件端口输入/输出宏汇编语句。
33 #include <asm/system.h> // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
34
35 /*
36  * These are set up by the setup-routine at boot-time:
37  */
/*
  * 这些是设置子程序 setup 在引导启动系统时设置的参数:
  */
38
// 参见对 boot/setup.s 的注释, 和 setup 程序读取并保留的参数表。
39 #define ORIG_X          (*(unsigned char *)0x90000) // 光标列号。
40 #define ORIG_Y          (*(unsigned char *)0x90001) // 光标行号。
41 #define ORIG_VIDEO_PAGE (*(unsigned short *)0x90004) // 显示页面。
42 #define ORIG_VIDEO_MODE ((*(unsigned short *)0x90006) & 0xff) // 显示模式。
43 #define ORIG_VIDEO_COLS (((*(unsigned short *)0x90006) & 0xff00) >> 8) // 字符列数。
44 #define ORIG_VIDEO_LINES (25) // 显示行数。
45 #define ORIG_VIDEO_EGA_AX (*(unsigned short *)0x90008) // [??]
46 #define ORIG_VIDEO_EGA_BX (*(unsigned short *)0x9000a) // 显示内存大小和色彩模式。
47 #define ORIG_VIDEO_EGA_CX (*(unsigned short *)0x9000c) // 显示卡特性参数。
48
// 定义显示器单色/彩色显示模式类型符号常数。
49 #define VIDEO_TYPE_MDA 0x10 /* Monochrome Text Display */ /* 单色文本 */
50 #define VIDEO_TYPE_CGA 0x11 /* CGA Display */ /* CGA 显示器 */
51 #define VIDEO_TYPE_EGAM 0x20 /* EGA/VGA in Monochrome Mode */ /* EGA/VGA 单色 */
52 #define VIDEO_TYPE_EGAC 0x21 /* EGA/VGA in Color Mode */ /* EGA/VGA 彩色 */
53
54 #define NPAR 16
55
56 extern void keyboard_interrupt(void); // 键盘中断处理程序(keyboard.S)。
57
58 static unsigned char video_type; /* Type of display being used */

```

```

59 static unsigned long    video_num_columns;    /* 使用的显示类型 */
60 static unsigned long    video_size_row;        /* Number of text columns */
61 static unsigned long    video_num_lines;        /* Bytes per row */
62 static unsigned char    video_page;            /* Number of test lines */
63 static unsigned long    video_mem_start;        /* Initial video page */
64 static unsigned long    video_mem_end;          /* Start of video RAM */
65 static unsigned short   video_port_reg;        /* End of video RAM (sort of) */
66 static unsigned short   video_port_val;        /* Video register select port */
67 static unsigned short   video_erase_char;      /* Video register value port */
68                                     /* Char+Attrib to erase with */
69                                     /* 擦除字符属性与字符 (0x0720) */
70 // 以下这些变量用于屏幕滚屏操作。
71 static unsigned long    origin;                /* Used for EGA/VGA fast scroll */ // scr_start.
72 static unsigned long    scr_end;                /* Used for EGA/VGA fast scroll */
73 static unsigned long    pos;                    /* 用于 EGA/VGA 快速滚屏 */ // 滚屏起始内存地址。
74 static unsigned long    x,y;                    /* 用于 EGA/VGA 快速滚屏 */ // 滚屏末端内存地址。
75 static unsigned long    top,bottom;            // 当前光标对应的显示内存位置。
76 // state 用于标明处理 ESC 转义序列时的当前步骤。npar, par[] 用于存放 ESC 序列的中间处理参数。
77 static unsigned long    state=0;                // 当前光标位置。
78 static unsigned long    npar, par[NPAR];        // 滚动时顶行行号; 底行行号。
79 static unsigned long    ques=0;                // ANSI 转义字符序列处理状态。
80 static unsigned char    attr=0x07;              // ANSI 转义字符序列参数个数和参数数组。
81 static void sysbeep(void);                      // 字符属性(黑底白字)。
82                                     // 系统蜂鸣函数。
83 /*
84  * this is what the terminal answers to a ESC-Z or csi0c
85  * query (= vt100 response).
86  */
87 /*
88  * 下面是终端回应 ESC-Z 或 csi0c 请求的应答(=vt100 响应)。
89  */
90 // csi - 控制序列引导码(Control Sequence Introducer)。
91 #define RESPONSE "\033[?1;2c"
92 /* NOTE! gotoxy thinks x==video_num_columns is ok */
93 /* 注意! gotoxy 函数认为 x==video_num_columns, 这是正确的 */
94 // 跟踪光标当前位置。
95 // 参数: new_x - 光标所在列号; new_y - 光标所在行号。
96 // 更新当前光标位置变量 x, y, 并修正 pos 指向光标在显示内存中的对应位置。
97 static inline void gotoxy(unsigned int new_x, unsigned int new_y)
98 {

```

```

// 如果输入的光标行号超出显示器列数，或者光标行号超出显示的最大行数，则退出。
90     if (new_x > video\_num\_columns || new_y >= video\_num\_lines)
91         return;
// 更新当前光标变量；更新光标位置对应的在显示内存中位置变量 pos。
92     x=new_x;
93     y=new_y;
94     pos=origin + y*video\_size\_row + (x<<1);
95 }
96
//// 设置滚屏起始显示内存地址。
97 static inline void set\_origin(void)
98 {
99     cli();
// 首先选择显示控制数据寄存器 r12，然后写入卷屏起始地址高字节。向右移动 9 位，表示向右移动
// 8 位，再除以 2(2 字节代表屏幕上 1 字符)。是相对于默认显示内存操作的。
100    outb\_p(12, video\_port\_reg);
101    outb\_p(0xff&((origin-video\_mem\_start)>>9), video\_port\_val);
// 再选择显示控制数据寄存器 r13，然后写入卷屏起始地址底字节。向右移动 1 位表示除以 2。
102    outb\_p(13, video\_port\_reg);
103    outb\_p(0xff&((origin-video\_mem\_start)>>1), video\_port\_val);
104    sti();
105 }
106
//// 向上卷动一行（屏幕窗口向下移动）。
// 将屏幕窗口向下移动一行。参见程序列表后说明。
107 static void scrup(void)
108 {
// 如果显示类型是 EGA，则执行以下操作。
109     if (video\_type == VIDEO\_TYPE\_EGAC || video\_type == VIDEO\_TYPE\_EGAM)
110     {
// 如果移动起始行 top=0，移动最底行 bottom=video_num_lines=25，则表示整屏窗口向下移动。
111         if (!top && bottom == video\_num\_lines) {
// 调整屏幕显示对应内存的起始位置指针 origin 为向下移一行屏幕字符对应的内存位置，同时也调整
// 当前光标对应的内存位置以及屏幕末行末端字符指针 scr_end 的位置。
112             origin += video\_size\_row;
113             pos += video\_size\_row;
114             scr\_end += video\_size\_row;
// 如果屏幕末端最后一个显示字符所对应的显示内存指针 scr_end 超出了实际显示内存的末端，则将
// 屏幕内容内存数据移动到显示内存的起始位置 video_mem_start 处，并在出现的新行上填入空格字符。
115             if (scr\_end > video\_mem\_end) {
// %0 - eax(擦除字符+属性)；%1 - ecx((显示器字符行数-1)所对应的字符数/2，是以长字移动)；
// %2 - edi(显示内存起始位置 video_mem_start)；%3 - esi(屏幕内容对应的内存起始位置 origin)。
// 移动方向：[edi]→[esi]，移动 ecx 个长字。
116                 __asm__("cld\n\t" // 清方向位。
117                     "rep\n\t" // 重复操作，将当前屏幕内存数据
118                     "movsl\n\t" // 移动到显示内存起始处。
119                     "movl \_video\_num\_columns,%1\n\t" // ecx=1 行字符数。
120                     "rep\n\t" // 在新行上填入空格字符。
121                     "stosw"
122                     ":: \"a\" \(<a href='\"#\"'>video\_erase\_char),
123                     \"c\" \(\(<a href='\"#\"'>video\_num\_lines-1)*<a href='\"#\"'>video_num_columns>>1),
124                     \"D\" \(<a href='\"#\"'>video\_mem\_start),
125                     \"S\" \(<a href='\"#\"'>origin)
```

```

126                                     : "cx", "di", "si");
// 根据屏幕内存数据移动后的情况, 重新调整当前屏幕对应内存的起始指针、光标位置指针和屏幕末端
// 对应内存指针 scr_end.
127                                     scr_end -= origin-video mem start;
128                                     pos -= origin-video mem start;
129                                     origin = video mem start;
130                                 } else {
// 如果调整后的屏幕末端对应的内存指针 scr_end 没有超出显示内存的末端 video_mem_end, 则只需在
// 新行上填入擦除字符(空格字符)。
// %0 - eax(擦除字符+属性); %1 - ecx(显示器字符行数); %2 - edi(屏幕对应内存最后一行开始处);
131                                     __asm__( "cld\n\t"          // 清方向位。
132                                             "rep\n\t"          // 重复操作, 在新出现行上
133                                             "stosw"          // 填入擦除字符(空格字符)。
134                                             :: "a" (video_erase_char),
135                                             "c" (video_num_columns),
136                                             "D" (scr_end-video_size_row)
137                                             : "cx", "di");
138                                 }
// 向显示控制器中写入新的屏幕内容对应的内存起始位置值。
139                                     set_origin();
// 则表示不是整屏移动。也即表示从指定行 top 开始的所有行向上移动 1 行(删除 1 行)。此时直接
// 将屏幕从指定行 top 到屏幕末端所有行对应的显示内存数据向上移动 1 行, 并在新出现的行上填入擦
// 除字符。
// %0-eax(擦除字符+属性); %1-ecx(top 行下 1 行开始到屏幕末行的行数所对应的内存长字数);
// %2-edi(top 行所处的内存位置); %3-esi(top+1 行所处的内存位置)。
140                                 } else {
141                                     __asm__( "cld\n\t"          // 清方向位。
142                                             "rep\n\t"          // 循环操作, 将 top+1 到 bottom 行
143                                             "movsl\n\t"          // 所对应的内存块移到 top 行开始处。
144                                             "movl _video_num_columns, %%ecx\n\t" // ecx = 1 行字符数。
145                                             "rep\n\t"          // 在新行上填入擦除字符。
146                                             "stosw"
147                                             :: "a" (video_erase_char),
148                                             "c" ((bottom-top-1)*video_num_columns>>1),
149                                             "D" (origin+video_size_row*top),
150                                             "S" (origin+video_size_row*(top+1))
151                                             : "cx", "di", "si");
152                                 }
153                             }
// 如果显示类型不是 EGA(是 MDA), 则执行下面移动操作。因为 MDA 显示控制卡会自动调整超出显示范
// 围的情况, 也即会自动翻卷指针, 所以这里不对屏幕内容对应内存超出显示内存的情况单独处理。处
// 理方法与 EGA 非整屏移动情况完全一样。
154                             else /* Not EGA/VGA */
155                             {
156                                 __asm__( "cld\n\t"
157                                         "rep\n\t"
158                                         "movsl\n\t"
159                                         "movl _video_num_columns, %%ecx\n\t"
160                                         "rep\n\t"
161                                         "stosw"
162                                         :: "a" (video_erase_char),
163                                         "c" ((bottom-top-1)*video_num_columns>>1),
164                                         "D" (origin+video_size_row*top),

```

```

165         "S" (origin+video_size_row*(top+1))
166         : "cx", "di", "si");
167     }
168 }
169
170 // 向下卷动一行（屏幕窗口向上移动）。
171 // 将屏幕窗口向上移动一行，屏幕显示的内容向下移动 1 行，在被移动开始行的上方出现一新行。参见
172 // 程序列表后说明。处理方法与 scrup() 相似，只是为了在移动显示内存数据时不出现数据覆盖错误情
173 // 况，复制是以反方向进行的，也即从屏幕倒数第 2 行的最后一个字符开始复制
174 static void scrdown(void)
175 {
176     // 如果显示类型是 EGA，则执行下列操作。
177     // [??好象 if 和 else 的操作完全一样啊！为什么还要分别处理呢？难道与任务切换有关？]
178     if (video_type == VIDEO_TYPE EGAC || video_type == VIDEO_TYPE EGAM)
179     {
180         // %0-eax(擦除字符+属性)；%1-ecx(top 行开始到屏幕末行-1 行的行数所对应的内存长字数)；
181         // %2-edi(屏幕右下角最后一个长字位置)；%3-esi(屏幕倒数第 2 行最后一个长字位置)。
182         // 移动方向：[esi]→[edi]，移动 ecx 个长字。
183         __asm__( "std\n\t" // 置方向位。
184                 "rep\n\t" // 重复操作，向下移动从 top 行到 bottom-1 行
185                 "movsl\n\t" // 对应的内存数据。
186                 "addl $2,%edi\n\t" /* %edi has been decremented by 4 */
187                                    /* %edi 已经减 4，因为也是方向填擦除字符 */
188                 "movl _video_num_columns,%ecx\n\t" // 置 ecx=1 行字符数。
189                 "rep\n\t" // 将擦除字符填入上方新行中。
190                 "stosw"
191                 :: "a" (video_erase_char),
192                   "c" ((bottom-top-1)*video_num_columns>>1),
193                   "D" (origin+video_size_row*bottom-4),
194                   "S" (origin+video_size_row*(bottom-1)-4)
195                 : "ax", "cx", "di", "si");
196     }
197     // 如果不是 EGA 显示类型，则执行以下操作（目前与上面完全一样）。
198     else /* Not EGA/VGA */
199     {
200         __asm__( "std\n\t"
201                 "rep\n\t"
202                 "movsl\n\t"
203                 "addl $2,%edi\n\t" /* %edi has been decremented by 4 */
204                                    /* %edi 已经减 4，因为也是方向填擦除字符 */
205                 "movl _video_num_columns,%ecx\n\t"
206                 "rep\n\t"
207                 "stosw"
208                 :: "a" (video_erase_char),
209                   "c" ((bottom-top-1)*video_num_columns>>1),
210                   "D" (origin+video_size_row*bottom-4),
211                   "S" (origin+video_size_row*(bottom-1)-4)
212                 : "ax", "cx", "di", "si");
213     }
214 }
215
216 // 光标位置下移一行(lf - line feed 换行)。
217 static void lf(void)
218 {

```



```

// 如果光标没有处在倒数第 2 行之后，则直接修改光标当前行变量 y++，并调整光标对应显示内存位置
// pos(加上屏幕一行字符所对应的内存长度)。
206     if (y+1<bottom) {
207         y++;
208         pos += video_size_row;
209         return;
210     }
// 否则需要将屏幕内容上移一行。
211     scrup();
212 }
213
//// 光标上移一行(ri - reverse line feed 反向换行)。
214 static void ri(void)
215 {
// 如果光标不在第 1 行上，则直接修改光标当前行标量 y--，并调整光标对应显示内存位置 pos，减去
// 屏幕上一行字符所对应的内存长度字节数。
216     if (y>top) {
217         y--;
218         pos -= video_size_row;
219         return;
220     }
// 否则需要将屏幕内容下移一行。
221     scrdown();
222 }
223
// 光标回到第 1 列(0 列)左端(cr - carriage return 回车)。
224 static void cr(void)
225 {
// 光标所在的列号*2 即 0 列到光标所在列对应的内存字节长度。
226     pos -= x<<1;
227     x=0;
228 }
229
// 擦除光标前一字符(用空格替代)(del - delete 删除)。
230 static void del(void)
231 {
// 如果光标没有处在 0 列，则将光标对应内存位置指针 pos 后退 2 字节(对应屏幕上一个字符)，然后
// 将当前光标变量列值减 1，并将光标所在位置字符擦除。
232     if (x) {
233         pos -= 2;
234         x--;
235         *(unsigned short *)pos = video_erase_char;
236     }
237 }
238
//// 删除屏幕上与光标位置相关的部分，以屏幕为单位。csi - 控制序列引导码(Control Sequence
// Introducer)。
// ANSI 转义序列：'ESC [sJ'(s = 0 删除光标到屏幕底端；1 删除屏幕开始到光标处；2 整屏删除)。
// 参数：par - 对应上面 s。
239 static void csi_J(int par)
240 {
241     long count __asm__("cx"); // 设为寄存器变量。
242     long start __asm__("di");

```

```

243 // 首先根据三种情况分别设置需要删除的字符数和删除开始的显示内存位置。
244 switch (par) {
245     case 0: /* erase from cursor to end of display */ /* 擦除光标到屏幕底端 */
246         count = (scr_end-pos)>>1;
247         start = pos;
248         break;
249     case 1: /* erase from start to cursor */ /* 删除从屏幕开始到光标处的字符 */
250         count = (pos-origin)>>1;
251         start = origin;
252         break;
253     case 2: /* erase whole display */ /* 删除整个屏幕上的字符 */
254         count = video_num_columns * video_num_lines;
255         start = origin;
256         break;
257     default:
258         return;
259 }
260 // 然后使用擦除字符填写删除字符的地方。
261 // %0 - ecx(要删除的字符数 count); %1 - edi(删除操作开始地址); %2 - eax(填入的擦除字符)。
262 __asm__( "cld\n\t"
263         "rep\n\t"
264         "stosw\n\t"
265         ":: \"c\" (count),
266         "D\" (start), \"a\" (video_erase_char)
267         : \"cx\", \"di\" );
268 }
269
270 // 删除行内与光标位置相关的部分，以一行为单位。
271 // ANSI 转义字符序列: 'ESC [sK' (s = 0 删除到行尾; 1 从开始删除; 2 整行都删除)。
272 static void csi_K(int par)
273 {
274     long count __asm__( "cx" ); // 设置寄存器变量。
275     long start __asm__( "di" );
276
277     // 首先根据三种情况分别设置需要删除的字符数和删除开始的显示内存位置。
278     switch (par) {
279         case 0: /* erase from cursor to end of line */ /* 删除光标到行尾字符 */
280             if (x>=video_num_columns)
281                 return;
282             count = video_num_columns-x;
283             start = pos;
284             break;
285         case 1: /* erase from start of line to cursor */ /* 删除从行开始到光标处 */
286             start = pos - (x<<1);
287             count = (x<video_num_columns)?x:video_num_columns;
288             break;
289         case 2: /* erase whole line */ /* 将整行字符全删除 */
290             start = pos - (x<<1);
291             count = video_num_columns;
292             break;
293     default:
294         return;

```

```

290     }
    // 然后使用擦除字符填写删除字符的地方。
    // %0 - ecx(要删除的字符数 count); %1 - edi(删除操作开始地址); %2 - eax(填入的擦除字符)。
291     __asm__( "cld\n\t"
292             "rep\n\t"
293             "stosw\n\t"
294             ":: \"c\" (count),
295             "D\" (start), \"a\" (video_erase_char)
296             : \"cx\", \"di\");
297 }
298
    ///// 允许翻译(重显) (允许重新设置字符显示方式, 比如加粗、加下划线、闪烁、反显等)。
    // ANSI 转义字符序列: 'ESC [nm'. n = 0 正常显示; 1 加粗; 4 加下划线; 7 反显; 27 正常显示。
299 void csi_m(void)
300 {
301     int i;
302
303     for (i=0; i<=npar; i++)
304         switch (par[i]) {
305             case 0: attr=0x07; break;
306             case 1: attr=0x0f; break;
307             case 4: attr=0x0f; break;
308             case 7: attr=0x70; break;
309             case 27: attr=0x07; break;
310         }
311 }
312
    ///// 根据设置显示光标。
    // 根据显示内存光标对应位置 pos, 设置显示控制器光标的显示位置。
313 static inline void set_cursor(void)
314 {
315     cli();
    // 首先使用索引寄存器端口选择显示控制数据寄存器 r14(光标当前显示位置高字节), 然后写入光标
    // 当前位置高字节(向右移动 9 位表示高字节移到低字节再除以 2)。是相对于默认显示内存操作的。
316     outb_p(14, video_port_reg);
317     outb_p(0xff & ((pos-video_mem_start)>>9), video_port_val);
    // 再使用索引寄存器选择 r15, 并将光标当前位置低字节写入其中。
318     outb_p(15, video_port_reg);
319     outb_p(0xff & ((pos-video_mem_start)>>1), video_port_val);
320     sti();
321 }
322
    ///// 发送对终端 VT100 的响应序列。
    // 将响应序列放入读缓冲队列中。
323 static void respond(struct tty_struct * tty)
324 {
325     char * p = RESPONSE;
326
327     cli(); // 关中断。
328     while (*p) { // 将字符序列放入写队列。
329         PUTCH(*p, tty->read_q);
330         p++;
331     }

```

```

332     sti();                // 开中断。
333     copy_to_cooked(tty);  // 转换成规范模式(放入辅助队列中)。
334 }
335
336     ///// 在光标处插入一空格字符。
337 static void insert_char(void)
338 {
339     int i=x;
340     unsigned short tmp, old = video_erase_char;
341     unsigned short * p = (unsigned short *) pos;
342
343     // 光标开始的所有字符右移一格，并将擦除字符插入在光标所在处。
344     // 若一行上都有字符的话，则行最后一个字符将不会更动☺?
345     while (i++<video_num_columns) {
346         tmp=*p;
347         *p=old;
348         old=tmp;
349         p++;
350     }
351
352     ///// 在光标处插入一行（则光标将处在新的空行上）。
353     // 将屏幕从光标所在行到屏幕底向下卷动一行。
354 static void insert_line(void)
355 {
356     int oldtop,oldbottom;
357
358     oldtop=top;                // 保存原 top, bottom 值。
359     oldbottom=bottom;
360     top=y;                    // 设置屏幕卷动开始行。
361     bottom = video_num_lines; // 设置屏幕卷动最后行。
362     scrdown();                // 从光标开始处，屏幕内容向下滚动一行。
363     top=oldtop;                // 恢复原 top, bottom 值。
364     bottom=oldbottom;
365 }
366
367     ///// 删除光标处的一个字符。
368 static void delete_char(void)
369 {
370     int i;
371     unsigned short * p = (unsigned short *) pos;
372
373     // 如果光标超出屏幕最右列，则返回。
374     if (x>=video_num_columns)
375         return;
376     // 从光标右一个字符开始到行末所有字符左移一格。
377     i = x;
378     while (++i < video_num_columns) {
379         *p = *(p+1);
380         p++;
381     }
382     // 最后一个字符处填入擦除字符(空格字符)。
383     *p = video_erase_char;

```

```

376 }
377
378 // 删除光标所在行。
379 // 从光标所在行开始屏幕内容上卷一行。
380 static void delete_line(void)
381 {
382     int oldtop, oldbottom;
383
384     oldtop=top; // 保存原 top, bottom 值。
385     oldbottom=bottom;
386     top=y; // 设置屏幕卷动开始行。
387     bottom = video_num_lines; // 设置屏幕卷动最后行。
388     scrup(); // 从光标开始处, 屏幕内容向上滚动一行。
389     top=oldtop; // 恢复原 top, bottom 值。
390     bottom=oldbottom;
391 }
392
393 // 在光标处插入 nr 个字符。
394 // ANSI 转义字符序列: 'ESC [n@'。
395 // 参数 nr = 上面 n。
396 static void csi_at(unsigned int nr)
397 {
398     // 如果插入的字符数大于一行字符数, 则截为一行字符数; 若插入字符数 nr 为 0, 则插入 1 个字符。
399     if (nr > video_num_columns)
400         nr = video_num_columns;
401     else if (!nr)
402         nr = 1;
403     // 循环插入指定的字符数。
404     while (nr--)
405         insert_char();
406 }
407
408 // 在光标位置处插入 nr 行。
409 // ANSI 转义字符序列'ESC [nL'。
410 static void csi_L(unsigned int nr)
411 {
412     // 如果插入的行数大于屏幕最多行数, 则截为屏幕显示行数; 若插入行数 nr 为 0, 则插入 1 行。
413     if (nr > video_num_lines)
414         nr = video_num_lines;
415     else if (!nr)
416         nr = 1;
417     // 循环插入指定行数 nr。
418     while (nr--)
419         insert_line();
420 }
421
422 // 删除光标处的 nr 个字符。
423 // ANSI 转义序列: 'ESC [nP'。
424 static void csi_P(unsigned int nr)
425 {
426     // 如果删除的字符数大于一行字符数, 则截为一行字符数; 若删除字符数 nr 为 0, 则删除 1 个字符。
427     if (nr > video_num_columns)
428         nr = video_num_columns;

```

```

415         else if (!nr)
416             nr = 1;
// 循环删除指定字符数 nr。
417         while (nr--)
418             delete\_char();
419     }
420
// 删除光标处的 nr 行。
// ANSI 转义序列: 'ESC [nM'。
421 static void csi\_M(unsigned int nr)
422 {
// 如果删除的行数大于屏幕最多行数, 则截为屏幕显示行数; 若删除的行数 nr 为 0, 则删除 1 行。
423     if (nr > video\_num\_lines)
424         nr = video\_num\_lines;
425     else if (!nr)
426         nr=1;
// 循环删除指定行数 nr。
427     while (nr--)
428         delete\_line();
429 }
430
431 static int saved\_x=0;           // 保存的光标列号。
432 static int saved\_y=0;           // 保存的光标行号。
433
// 保存当前光标位置。
434 static void save\_cur(void)
435 {
436     saved\_x=x;
437     saved\_y=y;
438 }
439
// 恢复保存的光标位置。
440 static void restore\_cur(void)
441 {
442     gotoxy(saved\_x, saved\_y);
443 }
444
// 控制台写函数。
// 从终端对应的 tty 写缓冲队列中取字符, 并显示在屏幕上。
445 void con\_write(struct tty\_struct * tty)
446 {
447     int nr;
448     char c;
449
// 首先取得写缓冲队列中现有字符数 nr, 然后针对每个字符进行处理。
450     nr = CHARS(tty->write_q);
451     while (nr--) {
// 从写队列中取一字符 c, 根据前面所处理字符的状态 state 分别处理。状态之间的转换关系为:
// state = 0: 初始状态; 或者原状态 4; 或者原状态 1, 但字符不是 '[';
//         1: 原状态 0, 并且字符是转义字符 ESC(0x1b = 033 = 27);
//         2: 原状态 1, 并且字符是 '[';
//         3: 原状态 2; 或者原状态 3, 并且字符是 ';' 或数字。
//         4: 原状态 3, 并且字符不是 ';' 或数字;

```

```

452         GETCH(tty->write_q, c);
453         switch(state) {
454             case 0:
455                 // 如果字符不是控制字符(c>31), 并且也不是扩展字符(c<127), 则
456                 if (c>31 && c<127) {
457                     // 若当前光标处在行末端或末端以外, 则将光标移到下行头列。并调整光标位置对应的内存指针 pos。
458                     if (x>video_num_columns) {
459                         x -= video_num_columns;
460                         pos -= video_size_row;
461                         lf();
462                     }
463                     // 将字符 c 写到显示内存中 pos 处, 并将光标右移 1 列, 同时也将 pos 对应地移动 2 个字节。
464                     __asm__ ("movb _attr, %%ah\n\t"
465                             "movw %%ax, %I\n\t"
466                             ":: \"a\" (c), \"m\" (*(short *)pos)
467                             : \"ax\");
468                     pos += 2;
469                     x++;
470                     // 如果字符 c 是转义字符 ESC, 则转换状态 state 到 1。
471                     } else if (c==27)
472                         state=1;
473                     // 如果字符 c 是换行符(10), 或是垂直制表符 VT(11), 或者是换页符 FF(12), 则移动光标到下一行。
474                     else if (c==10 || c==11 || c==12)
475                         lf();
476                     // 如果字符 c 是回车符 CR(13), 则将光标移动到头列(0 列)。
477                     else if (c==13)
478                         cr();
479                     // 如果字符 c 是 DEL(127), 则将光标右边一字符擦除(用空格字符替代), 并将光标移到被擦除位置。
480                     else if (c==ERASE_CHAR(tty))
481                         del();
482                     // 如果字符 c 是 BS(backspace, 8), 则将光标右移 1 格, 并相应调整光标对应内存位置指针 pos。
483                     else if (c==8) {
484                         if (x) {
485                             x--;
486                             pos -= 2;
487                         }
488                     }
489                     // 如果字符 c 是水平制表符 TAB(9), 则将光标移到 8 的倍数列上。若此时光标列数超出屏幕最大列数,
490                     // 则将光标移到下一行上。
491                     } else if (c==9) {
492                         c=8-(x&7);
493                         x += c;
494                         pos += c<<1;
495                         if (x>video_num_columns) {
496                             x -= video_num_columns;
497                             pos -= video_size_row;
498                             lf();
499                         }
500                         c=9;
501                     // 如果字符 c 是响铃符 BEL(7), 则调用蜂鸣函数, 是扬声器发声。
502                     } else if (c==7)
503                         sysbeep();
504                     break;
505                 // 如果原状态是 0, 并且字符是转义字符 ESC(0x1b = 033 = 27), 则转到状态 1 处理。

```



```

493             case 1:
494                 state=0;
495                 // 如果字符 c 是 '['，则将状态 state 转到 2。
496                 if (c=='[')
497                     state=2;
498                 // 如果字符 c 是 'E'，则光标移到下一行开始处(0 列)。
499                 else if (c=='E')
500                     gotoxy(0, y+1);
501                 // 如果字符 c 是 'M'，则光标上移一行。
502                 else if (c=='M')
503                     ri();
504                 // 如果字符 c 是 'D'，则光标下移一行。
505                 else if (c=='D')
506                     lf();
507                 // 如果字符 c 是 'Z'，则发送终端应答字符序列。
508                 else if (c=='Z')
509                     respond(tty);
510                 // 如果字符 c 是 '7'，则保存当前光标位置。注意这里代码写错！应该是(c=='7')。
511                 else if (x=='7')
512                     save_cur();
513                 // 如果字符 c 是 '8'，则恢复到原保存的光标位置。注意这里代码写错！应该是(c=='8')。
514                 else if (x=='8')
515                     restore_cur();
516                 break;
517             // 如果原状态是 1，并且上一字符是 '['，则转到状态 2 来处理。
518             case 2:
519                 // 首先对 ESC 转义字符序列参数使用的处理数组 par[] 清零，索引变量 npar 指向首项，并且设置状态
520                 // 为 3。若此时字符不是 '?'，则直接转到状态 3 去处理，否则去读一字符，再到状态 3 处理代码处。
521                 for(npar=0; npar<NPAR; npar++)
522                     par[npar]=0;
523                 npar=0;
524                 state=3;
525                 if (ques==(c=='?'))
526                     break;
527                 // 如果原来是状态 2；或者原来就是状态 3，但原字符是 ';' 或数字，则在下面处理。
528                 case 3:
529                 // 如果字符 c 是分号 ';'，并且数组 par 未满，则索引值加 1。
530                 if (c==';' && npar<NPAR-1) {
531                     npar++;
532                     break;
533                 // 如果字符 c 是数字字符 '0'-'9'，则将该字符转换成数值并与 npar 所索引的项组成 10 进制数。
534                 } else if (c>='0' && c<='9') {
535                     par[npar]=10*par[npar]+c-'0';
536                     break;
537                 // 否则转到状态 4。
538                 } else state=4;
539             // 如果原状态是状态 3，并且字符不是 ';' 或数字，则转到状态 4 处理。首先复位状态 state=0。
540             case 4:
541                 state=0;
542                 switch(c) {
543                 // 如果字符 c 是 'G' 或 '`'，则 par[] 中第一个参数代表列号。若列号不为零，则将光标右移一格。
544                 case 'G': case '`':
545                     if (par[0]) par[0]--;

```

```

530         gotoxy(par[0], y);
531         break;
// 如果字符 c 是'A'，则第一个参数代表光标上移的行数。若参数为 0 则上移一行。
532         case 'A':
533             if (!par[0]) par[0]++;
534             gotoxy(x, y-par[0]);
535             break;
// 如果字符 c 是'B'或'e'，则第一个参数代表光标下移的行数。若参数为 0 则下移一行。
536         case 'B': case 'e':
537             if (!par[0]) par[0]++;
538             gotoxy(x, y+par[0]);
539             break;
// 如果字符 c 是'C'或'a'，则第一个参数代表光标右移的格数。若参数为 0 则右移一格。
540         case 'C': case 'a':
541             if (!par[0]) par[0]++;
542             gotoxy(x+par[0], y);
543             break;
// 如果字符 c 是'D'，则第一个参数代表光标左移的格数。若参数为 0 则左移一格。
544         case 'D':
545             if (!par[0]) par[0]++;
546             gotoxy(x-par[0], y);
547             break;
// 如果字符 c 是'E'，则第一个参数代表光标向下移动的行数，并回到 0 列。若参数为 0 则下移一行。
548         case 'E':
549             if (!par[0]) par[0]++;
550             gotoxy(0, y+par[0]);
551             break;
// 如果字符 c 是'F'，则第一个参数代表光标向上移动的行数，并回到 0 列。若参数为 0 则上移一行。
552         case 'F':
553             if (!par[0]) par[0]++;
554             gotoxy(0, y-par[0]);
555             break;
// 如果字符 c 是'd'，则第一个参数代表光标所需的行号(从 0 计数)。
556         case 'd':
557             if (par[0]) par[0]--;
558             gotoxy(x, par[0]);
559             break;
// 如果字符 c 是'H'或'f'，则第一个参数代表光标移到的行号，第二个参数代表光标移到的列号。
560         case 'H': case 'f':
561             if (par[0]) par[0]--;
562             if (par[1]) par[1]--;
563             gotoxy(par[1], par[0]);
564             break;
// 如果字符 c 是'J'，则第一个参数代表以光标所处位置清屏的方式：
// ANSI 转义序列：'ESC [sJ' (s = 0 删除光标到屏幕底端；1 删除屏幕开始到光标处；2 整屏删除)。
565         case 'J':
566             csi_J(par[0]);
567             break;
// 如果字符 c 是'K'，则第一个参数代表以光标所在位置对行中字符进行删除处理的方式。
// ANSI 转义字符序列：'ESC [sK' (s = 0 删除到行尾；1 从开始删除；2 整行都删除)。
568         case 'K':
569             csi_K(par[0]);
570             break;

```

```

// 如果字符 c 是 'L'，表示在光标位置处插入 n 行 (ANSI 转义字符序列 'ESC [nL')。
571         case 'L':
572             csi\_L(par[0]);
573             break;
// 如果字符 c 是 'M'，表示在光标位置处删除 n 行 (ANSI 转义字符序列 'ESC [nM')。
574         case 'M':
575             csi\_M(par[0]);
576             break;
// 如果字符 c 是 'P'，表示在光标位置处删除 n 个字符 (ANSI 转义字符序列 'ESC [nP')。
577         case 'P':
578             csi\_P(par[0]);
579             break;
// 如果字符 c 是 '@'，表示在光标位置处插入 n 个字符 (ANSI 转义字符序列 'ESC [n@')。
580         case '@':
581             csi\_at(par[0]);
582             break;
// 如果字符 c 是 'm'，表示改变光标处字符的显示属性，比如加粗、加下划线、闪烁、反显等。
// ANSI 转义字符序列: 'ESC [nm'。n = 0 正常显示; 1 加粗; 4 加下划线; 7 反显; 27 正常显示。
583         case 'm':
584             csi\_m();
585             break;
// 如果字符 c 是 'r'，则表示用两个参数设置滚屏的起始行号和终止行号。
586         case 'r':
587             if (par[0]) par[0]--;
588             if (!par[1]) par[1] = video\_num\_lines;
589             if (par[0] < par[1] &&
590                 par[1] <= video\_num\_lines) {
591                 top=par[0];
592                 bottom=par[1];
593             }
594             break;
// 如果字符 c 是 's'，则表示保存当前光标所在位置。
595         case 's':
596             save\_cur();
597             break;
// 如果字符 c 是 'u'，则表示恢复光标到原保存的位置处。
598         case 'u':
599             restore\_cur();
600             break;
601     }
602 }
603 }
// 最后根据上面设置的光标位置，向显示控制器发送光标显示位置。
604     set\_cursor();
605 }
606
607 /*
608  * void con_init(void);
609  *
610  * This routine initializes console interrupts, and does nothing
611  * else. If you want the screen to clear, call tty_write with
612  * the appropriate escape-sequence.
613  */

```

```

614 * Reads the information preserved by setup.s to determine the current display
615 * type and sets everything accordingly.
616 */
/*
* void con_init(void);
* 这个子程序初始化控制台中断，其它什么都不做。如果你想让屏幕干净的话，就使用
* 适当的转义字符序列调用 tty_write() 函数。
*
* 读取 setup.s 程序保存的信息，用以确定当前显示器类型，并且设置所有相关参数。
*/
617 void con_init(void)
618 {
619     register unsigned char a;
620     char *display_desc = "????";
621     char *display_ptr;
622
623     video_num_columns = ORIG_VIDEO_COLS;    // 显示器显示字符列数。
624     video_size_row = video_num_columns * 2; // 每行需使用字节数。
625     video_num_lines = ORIG_VIDEO_LINES;    // 显示器显示字符行数。
626     video_page = ORIG_VIDEO_PAGE;          // 当前显示页面。
627     video_erase_char = 0x0720;             // 擦除字符(0x20 显示字符， 0x07 是属性)。
628
    // 如果原始显示模式等于 7，则表示是单色显示器。
629     if (ORIG_VIDEO_MODE == 7)               /* Is this a monochrome display? */
630     {
631         video_mem_start = 0xb0000;          // 设置单显映象内存起始地址。
632         video_port_reg = 0x3b4;            // 设置单显索引寄存器端口。
633         video_port_val = 0x3b5;            // 设置单显数据寄存器端口。
        // 根据 BIOS 中断 int 0x10 功能 0x12 获得的显示模式信息，判断显示卡单色显示卡还是彩色显示卡。
        // 如果使用上述中断功能所得到的 BX 寄存器返回值不等于 0x10，则说明是 EGA 卡。因此初始
        // 显示类型为 EGA 单色；所使用映象内存末端地址为 0xb8000；并置显示器描述字符串为 'EGAm'。
        // 在系统初始化期间显示器描述字符串将显示在屏幕的右上角。
634         if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
635         {
636             video_type = VIDEO_TYPE_EGAM; // 设置显示类型(EGA 单色)。
637             video_mem_end = 0xb8000;       // 设置显示内存末端地址。
638             display_desc = "EGAm";        // 设置显示描述字符串。
639         }
        // 如果 BX 寄存器的值等于 0x10，则说明是单色显示卡 MDA。则设置相应参数。
640         else
641         {
642             video_type = VIDEO_TYPE_MDA; // 设置显示类型(MDA 单色)。
643             video_mem_end = 0xb2000;     // 设置显示内存末端地址。
644             display_desc = "MDA";        // 设置显示描述字符串。
645         }
646     }
    // 如果显示模式不为 7，则为彩色模式。此时所用的显示内存起始地址为 0xb800；显示控制索引寄存
    // 器端口地址为 0x3d4；数据寄存器端口地址为 0x3d5。
647     else /* If not, it is color. */
648     {
649         video_mem_start = 0xb8000;        // 显示内存起始地址。
650         video_port_reg = 0x3d4;           // 设置彩色显示索引寄存器端口。
651         video_port_val = 0x3d5;           // 设置彩色显示数据寄存器端口。

```

```

// 再判断显卡类别。如果 BX 不等于 0x10, 则说明是 EGA 显卡。
652         if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
653         {
654             video_type = VIDEO_TYPE_EGAC; // 设置显示类型(EGA 彩色)。
655             video_mem_end = 0xbc000; // 设置显示内存末端地址。
656             display_desc = "EGAc"; // 设置显示描述字符串。
657         }
// 如果 BX 寄存器的值等于 0x10, 则说明是 CGA 显卡。则设置相应参数。
658         else
659         {
660             video_type = VIDEO_TYPE_CGA; // 设置显示类型(CGA)。
661             video_mem_end = 0xba000; // 设置显示内存末端地址。
662             display_desc = "CGA"; // 设置显示描述字符串。
663         }
664     }
665
666     /* Let the user know what kind of display driver we are using */
/* 让用户知道我们正在使用哪一类显示驱动程序 */
667
// 在屏幕的右上角显示显示描述字符串。采用的方法是直接将字符串写到显示内存的相应位置处。
// 首先将显示指针 display_ptr 指到屏幕第一行右端差 4 个字符处(每个字符需 2 个字节, 因此减 8)。
668     display_ptr = ((char *)video_mem_start) + video_size_row - 8;
// 然后循环复制字符串中的字符, 并且每复制一个字符都空开一个属性字节。
669     while (*display_desc)
670     {
671         *display_ptr++ = *display_desc++; // 复制字符。
672         display_ptr++; // 空开属性字节位置。
673     }
674
675     /* Initialize the variables used for scrolling (mostly EGA/VGA) */
/* 初始化用于滚屏的变量(主要用于 EGA/VGA) */
676
677     origin = video_mem_start; // 滚屏起始显示内存地址。
678     scr_end = video_mem_start + video_num_lines * video_size_row; // 滚屏结束内存地址。
679     top = 0; // 最顶行号。
680     bottom = video_num_lines; // 最底行号。
681
682     gotoxy(ORIG_X, ORIG_Y); // 初始化光标位置 x, y 和对应的内存位置 pos。
683     set_trap_gate(0x21, &keyboard_interrupt); // 设置键盘中断陷阱门。
684     outb_p(inb_p(0x21) & 0xfd, 0x21); // 取消 8259A 中对键盘中断的屏蔽, 允许 IRQ1。
685     a = inb_p(0x61); // 延迟读取键盘端口 0x61 (8255A 端口 PB)。
686     outb_p(a | 0x80, 0x61); // 设置禁止键盘工作(位 7 置位),
687     outb(a, 0x61); // 再允许键盘工作, 用以复位键盘操作。
688 }
689 /* from bsd-net-2: */
690
691 //// 停止蜂鸣。
// 复位 8255A PB 端口的位 1 和位 0。
691 void sysbeepstop(void)
692 {
693     /* disable counter 2 */ /* 禁止定时器 2 */
694     outb(inb_p(0x61) & 0xFC, 0x61);
695 }

```

```

696
697 int beepcount = 0;
698
// 开通蜂鸣。
// 8255A 芯片 PB 端口的位 1 用作扬声器的开门信号；位 0 用作 8253 定时器 2 的门信号，该定时器的
// 输出脉冲送往扬声器，作为扬声器发声的频率。因此要使扬声器蜂鸣，需要两步：首先开启 PB 端口
// 位 1 和位 0（置位），然后设置定时器发送一定的定时频率即可。
699 static void sysbeep(void)
700 {
701     /* enable counter 2 */ /* 开启定时器 2 */
702     outb_p(inb_p(0x61)|3, 0x61);
703     /* set command for counter 2, 2 byte write */ /* 送设置定时器 2 命令 */
704     outb_p(0xB6, 0x43);
705     /* send 0x637 for 750 HZ */ /* 设置频率为 750HZ，因此送定时值 0x637 */
706     outb_p(0x37, 0x42);
707     outb(0x06, 0x42);
708     /* 1/8 second */ /* 蜂鸣时间为 1/8 秒 */
709     beepcount = HZ/8;
710 }
711

```

7.5.3 其它信息

7.5.3.1 显示控制卡编程

这里仅给出和说明兼容显示卡端口的说明。描述了 MDA、CGA、EGA 和 VGA 显示控制卡的通用编程端口，这些端口都是与 CGA 使用的 MC6845 芯片兼容，其名称和用途见表 7-5 所示。其中以 CGA/EGA/VGA 的端口(0x3d0-0x3df)为例进行说明，MDA 的端口是 0x3b0 - 0x3bf。

对显示控制卡进行编程的基本步骤是：首先写显示卡的索引寄存器，选择要进行设置的显示控制内部寄存器之一(r0-r17)，然后将参数写到其数据寄存器端口。也即显示卡的数据寄存器端口每次只能对显示卡中的一个内部寄存器进行操作。内部寄存器见表 7-6 所示。

表 7-5 CGA 端口寄存器名称及作用

端口	读/写	名称和用途
0x3d4	写	CRT(6845)索引寄存器。用于选择通过端口 0x3b5 访问的各个数据寄存器(r0-r17)。
0x3d5	写	CRT(6845)数据寄存器。其中数据寄存器 r12-r15 还可以读。 各个数据寄存器的功能说明见下表。
0x3d8	读/写	模式控制寄存器。 位 7-6 未用； 位 5=1 允许闪烁； 位 4=1 640*200 图形模式； 位 3=1 允许视频； 位 2=1 单色显示； 位 1=1 图形模式；=0 文本模式； 位 0=1 80*25 文本模式；=0 40*25 文本模式。
0x3d9	读/写	CGA 调色板寄存器。选择所采用的色彩。 位 7-6 未用；

		位 5=1 激活色彩集：青(cyan)、紫(magenta)、白(white); =0 激活色彩集：红(red)、绿(green)、蓝(blue); 位 4=1 增强显示图形、文本背景色彩; 位 3=1 增强显示 40*25 的边框、320*200 的背景、640*200 的前景颜色; 位 2=1 显示红色：40*25 的边框、320*200 的背景、640*200 的前景; 位 1=1 显示绿色：40*25 的边框、320*200 的背景、640*200 的前景; 位 0=1 显示蓝色：40*25 的边框、320*200 的背景、640*200 的前景;
0x3da	读	CGA 显示状态寄存器。 位 7-4 未用; 位 3=1 在垂直回扫阶段; 位 2=1 光笔开关关闭; =0 光笔开关接通; 位 1=1 光笔选通有效; 位 0=1 可以不干扰显示访问显示内存; =0 此时不要使用显示内存。
0x3db	写	清除光笔锁存 (复位光笔寄存器)。
0x3dc	读/写	预设置光笔锁存 (强制光笔选通有效)。

表 7-6 MC6845 内部数据寄存器及初始值

编号	名称	单位	读/写	40*25 模式	80*25 模式	图形模式
r0	水平字符总数	字符	写	0x38	0x71	0x38
r1	水平显示字符数	字符	写	0x28	0x50	0x28
r2	水平同步位置	字符	写	0x2d	0x5a	0x2d
r3	水平同步脉冲宽度	字符	写	0x0a	0x0a	0x0a
r4	垂直字符总数	字符行	写	0x1f	0x1f	0x7f
r5	垂直同步脉冲宽度	扫描行	写	0x06	0x06	0x06
r6	垂直显示字符数	字符行	写	0x19	0x19	0x64
r7	垂直同步位置	字符行	写	0x1c	0x1c	0x70
r8	隔行/逐行选择		写	0x02	0x02	0x02
r9	最大扫描行数	扫描行	写	0x07	0x07	0x01
r10	光标开始位置	扫描行	写	0x06	0x06	0x06
r11	光标结束位置	扫描行	写	0x07	0x07	0x07
r12	显示内存起始位置(高)		写	0x00	0x00	0x00
r13	显示内存末端位置(低)		写	0x00	0x00	0x00
r14	光标当前位置(高)		读/写	可变		
r15	光标当前位置(低)		读/写			
r16	光笔当前位置(高)		读	可变		
r17	光笔当前位置(低)		读			

7.5.3.2 滚屏操作原理

滚屏操作是指将指定开始行和结束行的一块文本内容向上移动(向上卷动 scroll up)或向下移动(向下卷动 scroll down), 如果将屏幕看作是显示内存上对应屏幕内容的一个窗口的话, 那么将屏幕内容向上移即是窗口沿显示内存向下移动; 将屏幕内容向下移动即是窗口向下移动。在程序中就是重新设置显示控制器中显示内存的起始位置 origin 以及调整程序中相应的变量。对于这两种操作各自都有两种情况。

对于向上卷动, 当屏幕对应的显示内存窗口在向下移动后仍然在显示内存范围之内, 也即对应当前屏幕的内存块位置始终在显示内存起始位置(video_mem_start)和末端位置 video_mem_end 之间, 那么只需要调整显示控制器中起始显示内存位置即可。但是当对应屏幕的内存块位置在向下移动时超出了实际显示内存的末端(video_mem_end)这种情况, 就需要移动对应显示内存中的数据, 以保证所有当前屏幕数据都落在显示内存范围内。在这第二中情况, 程序中是将屏幕对应的内存数据移动到实际显示内存的开始位置处(video_mem_start)。

程序中实际的处理过程分三步进行。首先调整屏幕显示起始位置 origin; 然后判断对应屏幕内存数据是否超出显示内存下界(video_mem_end), 如果超出就将屏幕对应的内存数据移动到实际显示内存的开始位置处(video_mem_start); 最后对移动后屏幕上出现的新行用空格字符填满。见图 7-7 所示。其中图(a)对应第一种简单情况, 图(b)对应需要移动内存数据时的情况。

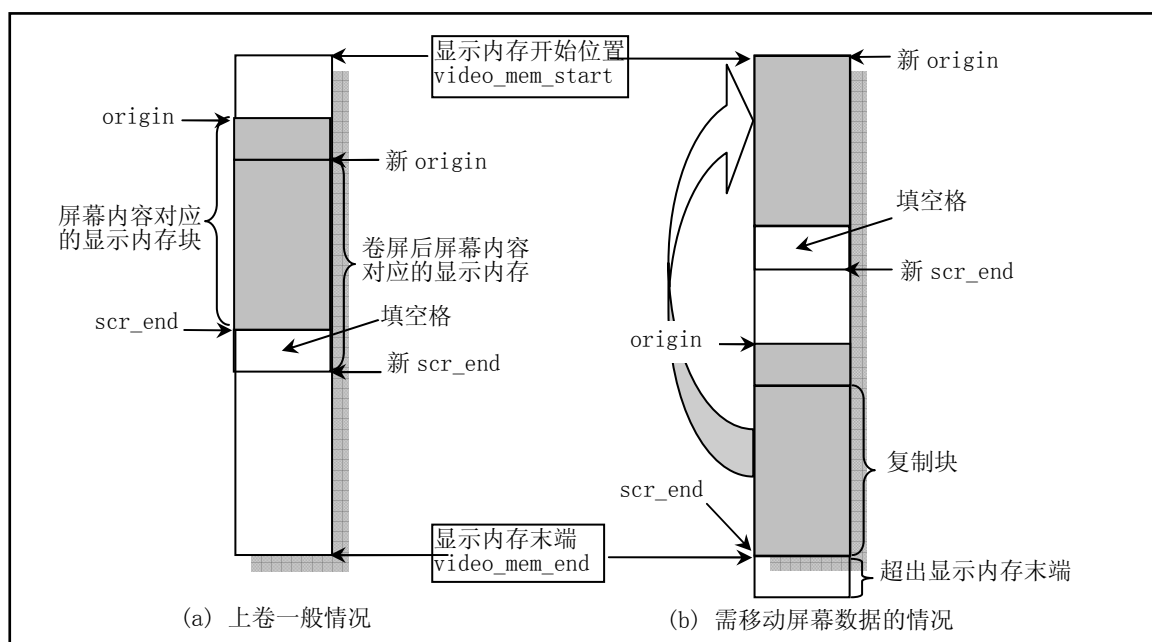


图 7-7 向上卷屏(scroll up)操作示意图

向下卷动屏幕的操作与向上卷屏相似, 也会遇到这两种类似情况, 只是由于屏幕窗口上移, 因此会在屏幕上方出现一空行, 并且在屏幕内容所对应的内存超出显示内存范围时需要将屏幕数据内存块往下移动到显示内存的末端位置。

7.5.3.3 ANSI 转义控制序列

终端通常有两部分功能, 分别作为计算机信息的输入设备(键盘)和输出设置(显示器)。终端可有許多控制命令, 使得终端执行一定的操作而不是仅仅在屏幕上显示一个字符。使用这种方式, 计算机就可以命令终端执行移动光标、切换显示模式和响铃等操作。为了理解程序的执行处理过程, 下面对终端控制命令进行一些简单描述。首先说明控制字符和控制序列的含义。

控制字符是指 ASCII 码表开头的 32 个字符(0x00 - 0x1f 或 0-31)以及字符 DEL(0x7f 或 127), 参见附录中的 ASCII 码表。通常一个指定类型的终端都会采用其中的一个子集作为控制字符, 而其它的控制字符将不起作用。例如, 对于 VT100 终端所采用的控制字符见表 7-7 所示。

表 7-7 控制字符

控制字符	八进制	十六进制	采取的行动
NUL	000	0x00	在输入时忽略（不保存在输入缓冲中）。
ENQ	005	0x05	传送应答消息。
BEL	007	0x07	从键盘发声响。
BS	010	0x08	将光标移向左边一个字符位置处。若光标已经处在左边沿，则无动作。
HT	011	0x09	将光标移到下一个制表位。若右侧已经没有制表位，则移到右边缘处。
LF	012	0x0a	此代码导致一个回车或换行操作（见换行模式）。
VT	013	0x0b	作用如 LF。
FF	014	0x0c	作用如 LF。
CR	015	0x0d	将光标移到当前行的左边缘处。
SO	016	0x0e	使用由 SCS 控制序列设计的 G1 字符集。
SI	017	0x0f	选择 G0 字符集。由 ESC 序列选择。
XON	021	0x11	使终端重新进行传输。
XOFF	023	0x13	使中断除发送 XOFF 和 XON 以外，停止发送其它所有代码。
CAN	030	0x18	如果在控制序列期间发送，则序列不会执行而立刻终止。同时会显示出错字符。
SUB	032	0x1a	作用同 CAN。
ESC	033	0x1b	产生一个控制序列。
DEL	177	0x7f	在输入时忽略（不保存在输入缓冲中）。

控制序列已经由 ANSI(美国国家标准局 American National Standards Institute)制定为标准：X3.64-1977。控制序列是指由一些非控制字符构成的一个特殊字符序列，终端在收到这个序列时并不是将它们直接显示在屏幕上，而是采取一定的控制操作，比如，移动光标、删除字符、删除行、插入字符或插入行等操作。ANSI 控制序列由以下一些基本元素组成：

控制序列引入码(Control Sequence Introducer - CSI)：表示一个转移序列，提供辅助的控制并且本身是影响随后一系列连续字符含义解释的前缀。通常，一般 CSI 都使用 ESC [。

参数(Parameter)：零个或多个数字字符组成的一个数值。

数值参数(Numeric Parameter)：表示一个数的参数，使用 n 表示。

选择参数(Selective Parameter)：用于从一功能子集中选择一个子功能，一般用 s 表示。通常，具有多个选择参数的一个控制序列所产生的作用，如同分立的几个控制序列。例如：CSI sa F CSI sb F CSI sc F 的作用是与 CSI sa F CSI sb F CSI sc F 完全一样的。

参数字符串(Parameter String)：用分号';'隔开的参数字符串。

默认值(Default)：当没有明确指定一个值或者值是 0 的话，就会指定一个与功能相关的值。

最后字符(Final character)：用于结束一个转义或控制序列。

图 7-8 中是一个控制序列的例子：取消所有字符的属性，然后开启下划线和反显属性。ESC [0;4;7m

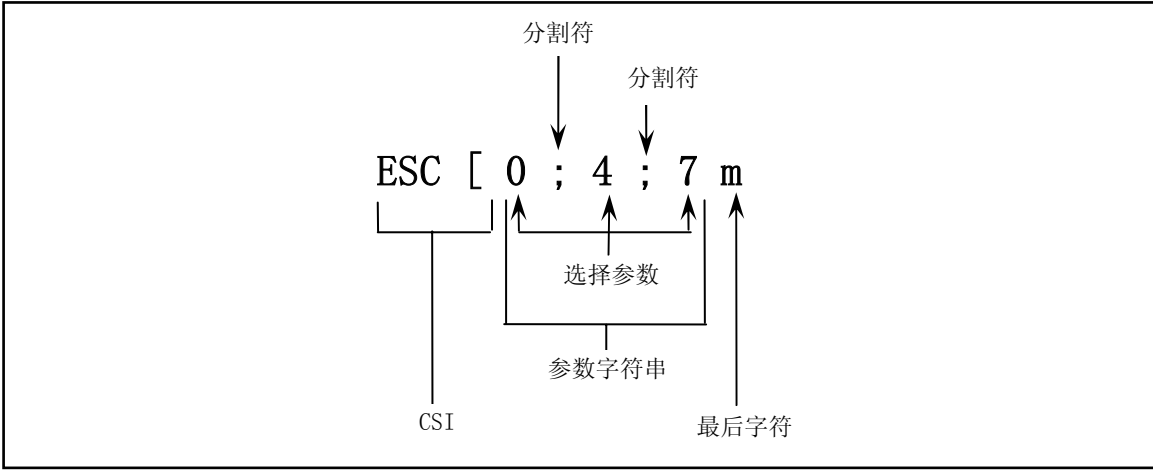


图 7-8 控制序列例子

表 7-8 是一些常用的控制序列列表。其中 E 表示 0x1b，如果 n 是 0 的话，则可以省略：E[0J == E[J

表 7-8 常用控制序列

转义序列	功能	转义序列	功能
E[nA	光标上移 n 行	E[nK	删除部分或整行：
E[nB	光标下移 n 行		n = 0 从光标处到行末端
E[nC	光标右移 n 个字符位置		n = 1 从行开始到光标处
E[nD	光标左移 n 个字符位置		n = 2 整行
E[n`	光标移动到字符 n 位置	E[nX	删除 n 个字符
E[na	光标右移 n 个字符位置	E[nS	向上卷屏 n 行（屏幕下移）
E[nd	光标移动到行 n 上	E[nT	向下卷屏 n 行（屏幕上移）
E[ne	光标下移 n 行	E[nm	设置字符显示属性：
E[nF	光标上移 n 行，停在行开始处		n = 0 普通属性（无属性）
E[nE	光标下移 n 行，停在行开始处		n = 1 粗（bold）
E[y;xH	光标移到 x,y 位置		n = 4 下划线（underscore）
E[H	光标移到屏幕左上角		n = 5 闪烁（blink）
E[y;xf	光标移到位置 x,y		n = 7 反显（reverse）
E[nZ	光标后移 n 制表位		n = 3X 设置前台显示色彩
E[nL	插入 n 条空白行		n = 4X 设置后台显示色彩
E[n@	插入 n 个空格字符		X = 0 黑 black X = 1 红 red
E[nM	删除 n 行		X = 2 绿 green X = 3 棕 brown
E[nP	删除 n 个字符		X = 4 蓝 blue X = 5 紫 magenta
E[nJ	擦除部分或全部显示字符：		X = 6 青 cyan X = 7 白 white
	n = 0 从光标处到屏幕底部；		使用分号可以同时设置多个属性，
	n = 1 从屏幕上端到光标处；		例如：E[0;1;33;40m
	n = 2 屏幕上所有字符。	E[s	保存光标位置
		E[u	恢复保存的光标位置

7.6 serial.c 程序

7.6.1 功能描述

本程序实现系统串行端口初始化，为使用串行终端设备作好准备工作。在 `rs_init()` 初始化函数中，设置了默认的串行通信参数，并设置串行端口的中断陷阱门（中断向量）。`rs_write()` 函数用于把串行终端设备写缓冲队列中的字符通过串行线路发送给远端的终端设备。

`rs_write()` 将在文件系统中用于操作字符设备文件时被调用。当一个程序往串行设备 `/dev/tty64` 文件执行写操作时，就会执行系统调用 `sys_write()`（在 `fs/read_write.c` 中），而这个系统调用在判别出所读文件是一个字符设备文件时，即会调用 `rw_char()` 函数（在 `fs/char_dev.c` 中），该函数则会根据所读设备的子设备号等信息，由字符设备读写函数表（设备开关表）调用 `rw_tty()`，最终调用到这里的串行终端写操作函数 `rs_write()`。

`rs_write()` 函数实际上只是开启串行发送保持寄存器已空中断标志，在 UART 将数据发送出去后允许发中断信号。具体发送操作是在 `rs_io.s` 程序中完成。

7.6.2 代码注释

程序 7-4 linux/kernel/chr_drv/serial.c

```

1  /*
2   *  linux/kernel/serial.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *      serial.c
9   *
10 *  This module implements the rs232 io functions
11 *      void rs_write(struct tty_struct * queue);
12 *      void rs_init(void);
13 *  and all interrupts pertaining to serial IO.
14 */
15 /*
16 *      serial.c
17 *  该程序用于实现 rs232 的输入输出功能
18 *      void rs_write(struct tty_struct *queue);
19 *      void rs_init(void);
20 *  以及与传输 IO 有关系的所有中断处理程序。
21 */
22
23 #include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
24 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
25                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
26 #include <asm/system.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
27 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
28
29 #define WAKEUP_CHARS (TTY_BUF_SIZE/4) // 当写队列中含有 WAKEUP_CHARS 个字符时，就开始发送。

```

```

23 extern void rs1\_interrupt(void);          // 串行口 1 的中断处理程序(rs_io.s, 34)。
24 extern void rs2\_interrupt(void);          // 串行口 2 的中断处理程序(rs_io.s, 38)。
25
    ///// 初始化串行端口
    // port: 串口 1 - 0x3F8, 串口 2 - 0x2F8。
26 static void init(int port)
27 {
28     outb\_p(0x80, port+3);    /* set DLAB of line control reg */
                                /* 设置线路控制寄存器的 DLAB 位(位 7) */
29     outb\_p(0x30, port);      /* LS of divisor (48 -> 2400 bps) */
                                /* 发送波特率因子低字节, 0x30->2400bps */
30     outb\_p(0x00, port+1);    /* MS of divisor */
                                /* 发送波特率因子高字节, 0x00 */
31     outb\_p(0x03, port+3);    /* reset DLAB */
                                /* 复位 DLAB 位, 数据位为 8 位 */
32     outb\_p(0x0b, port+4);    /* set DTR, RTS, OUT_2 */
                                /* 设置 DTR, RTS, 辅助用户输出 2 */
33     outb\_p(0x0d, port+1);    /* enable all intrs but writes */
                                /* 除了写(写保持空)以外, 允许所有中断源中断 */
34     (void) inb(port);        /* read data port to reset things (?) */
                                /* 读数据口, 以进行复位操作(?) */
35 }
36
    ///// 初始化串行中断程序和串行接口。
37 void rs\_init(void)
38 {
39     set\_intr\_gate(0x24, rs1\_interrupt);    // 设置串行口 1 的中断门向量(硬件 IRQ4 信号)。
40     set\_intr\_gate(0x23, rs2\_interrupt);    // 设置串行口 2 的中断门向量(硬件 IRQ3 信号)。
41     init(tty\_table[1].read_q.data);        // 初始化串行口 1(.data 是端口号)。
42     init(tty\_table[2].read_q.data);        // 初始化串行口 2。
43     outb(inb\_p(0x21)&0xE7, 0x21);          // 允许主 8259A 芯片的 IRQ3, IRQ4 中断信号请求。
44 }
45
46 /*
47  * This routine gets called when tty_write has put something into
48  * the write_queue. It must check wheter the queue is empty, and
49  * set the interrupt register accordingly
50  *
51  * void _rs_write(struct tty_struct * tty);
52  */
53
    // 在 tty_write() 已将数据放入输出(写)队列时会调用下面的子程序。必须首先
    // 检查写队列是否为空, 并相应设置中断寄存器。
    //
    ///// 串行数据发送输出。
    // 实际上只是开启串行发送保持寄存器已空中断标志, 在 UART 将数据发送出去后允许发中断信号。
53 void rs\_write(struct tty\_struct * tty)
54 {
55     cli();                          // 关中断。
    // 如果写队列不空, 则从 0x3f9(或 0x2f9) 首先读取中断允许寄存器内容, 添上发送保持寄存器中断
    // 允许标志(位 1)后, 再写回该寄存器。这样就会让串行设备由于要写(发送)字符而引发中断。
56     if (!EMPTY(tty->write_q))
57         outb(inb\_p(tty->write_q.data+1) | 0x02, tty->write_q.data+1);

```

```

58     sti();           // 开中断。
59 }
60

```

7.6.3 其它信息

7.6.3.1 异步串行通信芯片 UART

PC 微机的串行通信使用的异步串行通信芯片是 INS 8250 或 NS16450 兼容芯片，统称为 UART(通用异步接收发送器)。对 UART 的编程实际上是对其内部寄存器执行读写操作。因此可将 UART 看作是一组寄存器集合，包含发送、接收和控制三部分。UART 内部有 10 个寄存器，供 CPU 通过 IN/OUT 指令对其进行访问。这些寄存器的端口和用途见表 7-9 所示。其中端口 0x3f8-0x3fe 用于微机上 COM1 串行口，0x2f8-0x2fe 对应 COM2 端口。条件 DLAB(Divisor Latch Access Bit)是除数锁存访问位，是指线路控制寄存器的位 7。

表 7-9 UART 内部寄存器对应端口及用途

端口	读/写	条件	用途
0x3f8 (0x2f8)	写	DLAB=0	写发送保持寄存器。含有将发送的字符。
	读	DLAB=0	读接收缓存寄存器。含有收到的字符。
	读/写	DLAB=1	读/写波特率因子低字节 (LSB)。
0x3f9 (0x2f9)	读/写	DLAB=1	读/写波特率因子高字节 (MSB)。
	读/写	DLAB=0	读/写中断允许寄存器。 位 7-4 全 0 保留不用； 位 3=1 modem 状态中断允许； 位 2=1 接收器线路状态中断允许； 位 1=1 发送保持寄存器空中断允许； 位 0=1 已接收到数据中断允许。
0x3fa (0x2fa)	读		读中断标识寄存器。中断处理程序用以判断此次中断是 4 种中的那一种。 位 7-3 全 0 (不用)； 位 2-1 确定中断的优先级： = 11 接收状态有错中断，优先级最高； = 10 已接收到数据中断，优先级第 2； = 01 发送保持寄存器空中断，优先级第 3； = 00 modem 状态改变中断，优先级第 4。 位 0=0 有待处理中断；=1 无中断。
0x3fb (0x2fb)	写		写线路控制寄存器。 位 7=1 除数锁存访问位(DLAB)。 0 接收器，发送保持或中断允许寄存器访问； 位 6=1 允许间断； 位 5=1 保持奇偶位； 位 4=1 偶校验；=0 奇校验； 位 3=1 允许奇偶校验；=0 无奇偶校验； 位 2=1 1 位停止位；=0 无停止位； 位 1-0 数据位长度： = 00 5 位数据位；

			= 01 6 位数据位; = 10 7 位数据位; = 11 8 位数据位。
0x3fc (0x2fc)	写		写 modem 控制寄存器。 位 7-5 全 0 保留; 位 4=1 芯片处于循环反馈诊断操作模式; 位 3=1 辅助用户指定输出 2, 允许 INTRPT 到系统; 位 2=1 辅助用户指定输出 1, PC 机未用; 位 1=1 使请求发送 RTS 有效; 位 0=1 使数据终端就绪 DTR 有效。
0x3fd (0x2fd)	读		读线路状态寄存器。 位 7=0 保留; 位 6=1 发送移位寄存器为空; 位 5=1 发送保持寄存器为空, 可以取字符发送; 位 4=1 接收到满足中断条件的位序列; 位 3=1 帧格式错误; 位 2=1 奇偶校验错误; 位 1=1 超越覆盖错误; 位 0=1 接收器数据准备好, 系统可读取。
0x3fe (0x2fe)	读		读 modem 状态寄存器。δ 表示信号发生变化。 位 7=1 载波检测(CD)有效; 位 6=1 响铃指示(RI)有效; 位 5=1 数据设备就绪(DSR)有效; 位 4=1 清除发送 (CTS) 有效; 位 3=1 检测到 δ 载波; 位 2=1 检测到响铃信号边沿; 位 1=1 δ 数据设备就绪(DSR); 位 0=1 δ 清除发送(CTS)。

7.7 rs_io.s 程序

7.7.1 功能描述

该汇编程序实现 rs232 串行通信中断处理过程。在进行字符的传输和存储过程中, 该中断过程主要对终端的读、写缓冲队列进行操作。它把从串行线路上接收到的字符存入串行终端的读缓冲队列 read_q 中, 或把写缓冲队列 write_q 中需要发送出去的字符通过串行线路发送给远端的串行终端设备。

引起系统发生串行中断的情况有 4 种: a. 由于 modem 状态发生了变化; b. 由于线路状态发生了变化; c. 由于接收到字符; d. 由于在中断允许标志寄存器中设置了发送保持寄存器中断允许标志, 需要发送字符。对引起中断的前两种情况的处理过程是通过读取对应状态寄存器值, 从而使其复位。对于由于接收到字符的情况, 程序首先把该字符放入读缓冲队列 read_q 中, 然后调用 copy_to_cooked()函数转换成以字符行为单位的规范模式字符放入辅助队列 secondary 中。对于需要发送字符的情况, 则程序首先从写缓冲队列 write_q 尾指针处中取出一个字符发送出去, 再判断写缓冲队列是否已空, 若还有字符则

循环执行发送操作。

因此，在阅读本程序之前，最好先看一下 `include/linux/tty.h` 头文件。其中给出了字符缓冲队列的数据结构 `tty_queue`、终端的数据结构 `tty_struct` 和一些控制字符的值。另外还有一些对缓冲队列进行操作的宏定义。缓冲队列及其操作示意图请参见图 7-9 所示。

7.7.2 代码注释

程序 7-5 linux/kernel/chr_drv/rs_io.s

```

1  /*
2  *  linux/kernel/rs_io.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *      rs_io.s
9  *
10 * This module implements the rs232 io interrupts.
11 */
12 /*
13 * 该程序模块实现 rs232 输入输出中断处理程序。
14 */
15
16 // size 是读写队列缓冲区的字节长度。
17 size = 1024                                /* must be power of two !    必须是 2 的次方并且需
18                                         and must match the value 与 tty_io.c 中的值匹配!
19                                         in tty_io.c!!! */
20
21 /* these are the offsets into the read/write buffer structures */
22 /* 以下这些是读写缓冲结构中的偏移量 */
23 // 对应定义在 include/linux/tty.h 文件中 tty_queue 结构中各变量的偏移量。
24 rs_addr = 0                                // 串行端口号字段偏移（端口号是 0x3f8 或 0x2f8）。
25 head = 4                                  // 缓冲区中头指针字段偏移。
26 tail = 8                                  // 缓冲区中尾指针字段偏移。
27 proc_list = 12                            // 等待该缓冲的进程字段偏移。
28 buf = 16                                  // 缓冲区字段偏移。
29
30 startup = 256                             /* chars left in write queue when we restart it */
31                                         /* 当写队列里还剩 256 个字符空间(WAKEUP_CHARS)时，我们就可以写 */
32
33 /*
34 * These are the actual interrupt routines. They look where
35 * the interrupt is coming from, and take appropriate action.
36 */
37 /*
38 * 这些是实际的中断程序。程序首先检查中断的来源，然后执行相应
39 * 的处理。
40 */

```

```

33 .align 2
    /// 串行端口 1 中断处理程序入口点。
34 _rs1_interrupt:
35     pushl $_table_list+8    // tty 表中对应串口 1 的读写缓冲指针的地址入栈(tty_io.c, 99)。
36     jmp rs_int             // 字符缓冲队列结构格式请参见 include/linux/tty.h, 第 16 行。
37 .align 2
    /// 串行端口 2 中断处理程序入口点。
38 _rs2_interrupt:
39     pushl $_table_list+16   // tty 表中对应串口 2 的读写缓冲队列指针的地址入栈。
40 rs_int:
41     pushl %edx
42     pushl %ecx
43     pushl %ebx
44     pushl %eax
45     push %es
46     push %ds                /* as this is an interrupt, we cannot */
47     pushl $0x10            /* know that bs is ok. Load it */
48     pop %ds                /* 由于这是一个中断程序, 我们不知道 ds 是否正确, */
49     pushl $0x10            /* 所以加载它们(让 ds、es 指向内核数据段 */
50     pop %es
51     movl 24(%esp), %edx     // 将缓冲队列指针地址存入 edx 寄存器,
    // 也即上面 35 或 39 行上最先压入堆栈的地址。
52     movl (%edx), %edx      // 取读缓冲队列结构指针(地址)→edx。
    // 对于串行终端, data 字段存放着串行端口地址(端口号)。
53     movl rs_addr(%edx), %edx // 取串口 1 (或串口 2) 的端口号→edx。
54     addl $2, %edx          /* interrupt ident. reg */ /* edx 指向中断标识寄存器 */
55 rep_int:                  // 中断标识寄存器端口是 0x3fa (0x2fa), 参见上节列表后信息。
56     xorl %eax, %eax        // eax 清零。
57     inb %dx, %al           // 取中断标识字节, 用以判断中断来源(有 4 种中断情况)。
58     testb $1, %al         // 首先判断有无待处理的中断(位 0=1 无中断; =0 有中断)。
59     jne end                // 若无待处理中断, 则跳转至退出处理处 end。
60     cmpb $6, %al          /* this shouldn't happen, but ... */ /* 这不会发生, 但是... */
61     ja end                // al 值>6? 是则跳转至 end (没有这种状态)。
62     movl 24(%esp), %ecx    // 再取缓冲队列指针地址→ecx。
63     pushl %edx            // 将中断标识寄存器端口号 0x3fa(0x2fa)入栈。
64     subl $2, %edx         // 0x3f8(0x2f8)。
65     call jmp_table(,%eax,2) /* NOTE! not *4, bit0 is 0 already */ /* 不乘 4, 位 0 已是 0 */
    // 上面语句是指, 当有待处理中断时, al 中位 0=0, 位 2-1 是中断类型, 因此相当于已经将中断类型
    // 乘了 2, 这里再乘 2, 获得跳转表(第 79 行)对应各中断类型地址, 并跳转到那里去作相应处理。
    // 中断来源有 4 种: modem 状态发生变化; 要写(发送)字符; 要读(接收)字符; 线路状态发生变化。
    // 要发送字符中断是通过设置发送保持寄存器标志实现的。在 serial.c 程序中的 rs_write() 函数中,
    // 当写缓冲队列中有数据时, 就会修改中断允许寄存器内容, 添加上发送保持寄存器中断允许标志,
    // 从而在系统需要发送字符时引起串行中断发生。
66     popl %edx             // 弹出中断标识寄存器端口号 0x3fa (或 0x2fa)。
67     jmp rep_int           // 跳转, 继续判断有无待处理中断并继续处理。
68 end:    movb $0x20, %al   // 向中断控制器发送结束中断指令 EOI。
69     outb %al, $0x20       /* EOI */
70     pop %ds
71     pop %es
72     popl %eax
73     popl %ebx
74     popl %ecx
75     popl %edx

```

```

76      addl $4,%esp          # jump over _table_list entry # 丢弃缓冲队列指针地址。
77      iret
78
// 各中断类型处理程序地址跳转表，共有 4 种中断来源：
// modem 状态变化中断，写字符中断，读字符中断，线路状态有问题中断。
79 jmp_table:
80     .long modem_status,write_char,read_char,line_status
81
// 由于 modem 状态发生变化而引发此次中断。通过读 modem 状态寄存器对其进行复位操作。
82 .align 2
83 modem_status:
84     addl $6,%edx           /* clear intr by reading modem status reg */
85     inb %dx,%al           /* 通过读 modem 状态寄存器进行复位(0x3fe) */
86     ret
87
// 由于线路状态发生变化而引起这次串行中断。通过读线路状态寄存器对其进行复位操作。
88 .align 2
89 line_status:
90     addl $5,%edx           /* clear intr by reading line status reg. */
91     inb %dx,%al           /* 通过读线路状态寄存器进行复位(0x3fd) */
92     ret
93
// 由于串行设备（芯片）接收到字符而引起这次中断。将接收到的字符放到读缓冲队列 read_q 头
// 指针（head）处，并且让该指针前移一个字符位置。若 head 指针已经到达缓冲区末端，则让其
// 折返到缓冲区开始处。最后调用 C 函数 do_tty_interrupt()（也即 copy_to_cooked()），把读
// 入的字符经过一定处理放入规范模式缓冲队列（辅助缓冲队列 secondary）中。
94 .align 2
95 read_char:
96     inb %dx,%al           /* 读取字符→al。
97     movl %ecx,%edx         /* 当前串口缓冲队列指针地址→edx。
98     subl $_table_list,%edx // 缓冲队列指针表首址 - 当前串口队列指针地址→edx，
99     shr $3,%edx           // 差值/8。对于串口 1 是 1，对于串口 2 是 2。
100    movl (%ecx),%ecx       # read-queue # 取读缓冲队列结构地址→ecx。
101    movl head(%ecx),%ebx   // 取读队列中缓冲头指针→ebx。
102    movb %al,buf(%ecx,%ebx) // 将字符放在缓冲区中头指针所指的位置。
103    incl %ebx             // 将头指针前移一字节。
104    andl $size-1,%ebx     // 用缓冲区大小对头指针进行模操作。指针不能超过缓冲区大小。
105    cmpl tail(%ecx),%ebx  // 缓冲区头指针与尾指针比较。
106    je 1f                // 若相等，表示缓冲区满，跳转到标号 1 处。
107    movl %ebx,head(%ecx)  // 保存修改过的头指针。
108 1:    pushl %edx          // 将串口号压入堆栈(1- 串口 1, 2 - 串口 2)，作为参数，
109    call _do_tty_interrupt // 调用 tty 中断处理 C 函数（tty_io.c, 242, 145）。
110    addl $4,%esp          // 丢弃入栈参数，并返回。
111    ret
112
// 由于设置了发送保持寄存器允许中断标志而引起此次中断。说明对应串行终端的写字符缓冲队列中
// 有字符需要发送。于是计算出写队列中当前所含字符数，若字符数已小于 256 个则唤醒等待写操作
// 进程。然后从写缓冲队列尾部取出一个字符发送，并调整和保存尾指针。如果写缓冲队列已空，则
// 跳转到 write_buffer_empty 处处理写缓冲队列空的情况。
113 .align 2
114 write_char:
115     movl 4(%ecx),%ecx     # write-queue # 取写缓冲队列结构地址→ecx。
116     movl head(%ecx),%ebx // 取写队列头指针→ebx。

```

```

117      subl tail(%ecx),%ebx          // 头指针 - 尾指针 = 队列中字符数。
118      andl $size-1,%ebx            # nr chars in queue # 对指针取模运算。
119      je write_buffer_empty        // 如果头指针 = 尾指针，说明写队列无字符，跳转处理。
120      cmpl $startup,%ebx          // 队列中字符数超过 256 个？
121      ja 1f                       // 超过，则跳转处理。
122      movl proc_list(%ecx),%ebx    # wake up sleeping process # 唤醒等待的进程。
                                      // 取等待该队列的进程的指针，并判断是否为空。
123      testl %ebx,%ebx            # is there any? # 有等待的进程吗？
124      je 1f                       // 是空的，则向前跳转到标号 1 处。
125      movl $0, (%ebx)            // 否则将进程置为可运行状态(唤醒进程)。。
126 1:      movl tail(%ecx),%ebx      // 取尾指针。
127      movb buf(%ecx,%ebx),%al     // 从缓冲中尾指针处取一字符→al。
128      outb %al,%dx               // 向端口 0x3f8(0x2f8)送出到保持寄存器中。
129      incl %ebx                  // 尾指针前移。
130      andl $size-1,%ebx          // 尾指针若到缓冲区末端，则折回。
131      movl %ebx,tail(%ecx)        // 保存已修改过的尾指针。
132      cmpl head(%ecx),%ebx       // 尾指针与头指针比较，
133      je write_buffer_empty      // 若相等，表示队列已空，则跳转。
134      ret
// 处理写缓冲队列 write_q 已空的情况。若有等待写该串行终端的进程则唤醒之，然后屏蔽发送
// 保持寄存器空中断，不让发送保持寄存器空时产生中断。
135 .align 2
136 write_buffer_empty:
137      movl proc_list(%ecx),%ebx    # wake up sleeping process # 唤醒等待的进程。
                                      // 取等待该队列的进程的指针，并判断是否为空。
138      testl %ebx,%ebx            # is there any? # 有等待的进程吗？
139      je 1f                       # 无，则向前跳转到标号 1 处。
140      movl $0, (%ebx)            # 否则将进程置为可运行状态(唤醒进程)。
141 1:      incl %edx                # 指向端口 0x3f9(0x2f9)。
142      inb %dx,%al                # 读取中断允许寄存器。
143      jmp 1f                      # 稍作延迟。
144 1:      jmp 1f
145 1:      andb $0xd,%al           /* disable transmit interrupt */
                                      /* 屏蔽发送保持寄存器空中断(位 1) */
146      outb %al,%dx               // 写入 0x3f9(0x2f9)。
147      ret

```

7.8 tty_io.c 程序

7.8.1 功能描述

每个 tty 设备有 3 个缓冲队列，分别是读缓冲队列（read_q）、写缓冲队列（write_q）和辅助缓冲队列（secondary），定义在 tty_struct 结构中（include/linux/tty.h）。对于每个缓冲队列，读操作是从缓冲队列的左端取字符，并且把缓冲队列尾（tail）指针向右移动。而写操作则是往缓冲队列的右端添加字符，并且也把头(head)指针向右移动。这两个指针中，任何一个若移动到超出了缓冲队列的末端，则折回到左端重新开始。见图 7-9 所示。

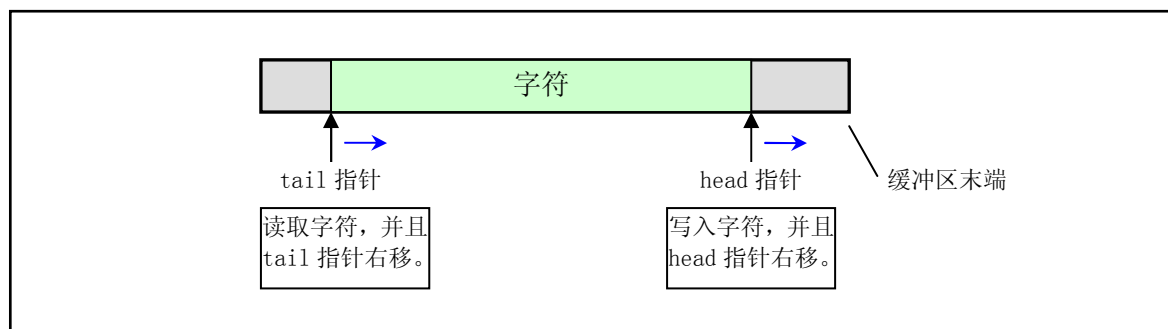


图 7-9 tty 字符缓冲队列的操作方式

本程序包括字符设备的上层接口函数。主要含有终端读/写函数 `tty_read()` 和 `tty_write()`。读操作的行规则函数 `copy_to_cooked()` 也在这里实现。

`tty_read()` 和 `tty_write()` 将在文件系统中用于操作字符设备文件时被调用。例如当一个程序读 `/dev/tty` 文件时, 就会执行系统调用 `sys_read()` (在 `fs/read_write.c` 中), 而这个系统调用在判别出所读文件是一个字符设备文件时, 即会调用 `rw_char()` 函数 (在 `fs/char_dev.c` 中), 该函数则会根据所读设备的子设备号等信息, 由字符设备读写函数表 (设备开关表) 调用 `rw_tty()`, 最终调用到这里的终端读操作函数 `tty_read()`。

`copy_to_cooked()` 函数由键盘中断过程调用 (通过 `do_tty_interrupt()`), 用于根据终端 `termios` 结构中设置的字符输入/输出标志 (例如 `INLCR`、`OUCLC`) 对 `read_q` 队列中的字符进行处理, 把字符转换成以字符行为单位的规范模式字符序列, 并保存在辅助字符缓冲队列 (规范模式缓冲队列) (`secondary`) 中, 供上述 `tty_read()` 读取。在转换处理期间, 若终端的回显标志 `L_ECHO` 置位, 则还会把字符放入写队列 `write_q` 中, 并调用终端写函数把该字符显示在屏幕上。如果是串行终端, 那么写函数将是 `rs_write()` (在 `serial.c`, 53 行)。`rs_write()` 会把串行终端写队列中的字符通过串行线路发送给串行终端, 并显示在串行终端的屏幕上。`copy_to_cooked()` 函数最后还将唤醒等待着辅助缓冲队列的进程。函数实现的步骤如下所示:

1. 如果读队列空或者辅助队列已经满, 则跳转到最后一步 (第 10 步), 否则执行以下操作;
2. 从读队列 `read_q` 的尾指针处取一字符, 并且尾指针前移一字符位置;
3. 若是回车 (CR) 或换行 (NL) 字符, 则根据终端 `termios` 结构中输入标志 (`ICRNL`、`INLCR`、`INOCR`) 的状态, 对该字符作相应转换。例如, 如果读取的是一个回车字符并且 `ICRNL` 标志是置位的, 则把它替换成换行字符;
4. 若大写转小写标志 `IUCLC` 是置位的, 则把字符替换成对应的小写字符;
5. 若规范模式标志 `ICANON` 是置位的, 则对该字符进行规范模式处理:
 - a. 若是删行字符 (^U), 则删除 `secondary` 中的一行字符 (队列头指针后退, 直到遇到回车或换行或队列已空为止);
 - b. 若是擦除字符 (^H), 则删除 `secondary` 中头指针处的一个字符, 头指针后退一个字符位置;
 - c. 若是停止字符 (^S), 则设置终端的停止标志 `stopped=1`;
 - d. 若是开始字符 (^Q), 则复位终端的停止标志。
6. 如果接收键盘信号标志 `ISIG` 是置位的, 则为进程生成对应键入控制字符的信号;
7. 如果是行结束字符 (例如 NL 或 ^D), 则辅助队列 `secondary` 的行数统计值 `data` 增 1;
8. 如果本地回显标志是置位的, 则把字符也放入写队列 `write_q` 中, 并调用终端写函数在屏幕上显示该字符;
9. 把该字符放入辅助队列 `secondary` 中, 返回上面第 1 步继续循环处理读队列中其它字符;
10. 最后唤醒睡眠在辅助队列上的进程。

在阅读下面程序时不免首先查看一下 `include/linux/tty.h` 头文件。在该头文件定义了 `tty` 字符缓冲队列

的数据结构以及一些宏操作定义。另外还定义了控制字符的 ASCII 码值。

7.8.2 代码注释

程序 7-6 linux/kernel/chr_drv/tty_io.c

```

1  /*
2  *  linux/kernel/tty_io.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  'tty_io.c' gives an orthogonal feeling to tty's, be they consoles
9  *  or rs-channels. It also implements echoing, cooked mode etc.
10 *
11 *  Kill-line thanks to John T Kohl.
12 */
13 /*
14 *  'tty_io.c' 给 tty 一种非相关的感觉，是控制台还是串行通道。该程序同样
15 *  实现了回显、规范(熟)模式等。
16 *
17 *  Kill-line, 谢谢 John T Kahl.
18 */
19 #include <ctype.h>          // 字符类型头文件。定义了一些有关字符类型判断和转换的宏。
20 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
21 #include <signal.h>         // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
22
23 // 下面给出相应信号在信号位图中的对应比特位。
24 #define ALRMASK (1<<(SIGALRM-1)) // 警告(alarm)信号屏蔽位。
25 #define KILLMASK (1<<(SIGKILL-1)) // 终止(kill)信号屏蔽位。
26 #define INTMASK (1<<(SIGINT-1)) // 键盘中断(int)信号屏蔽位。
27 #define QUITMASK (1<<(SIGQUIT-1)) // 键盘退出(quit)信号屏蔽位。
28 #define TSTPMASK (1<<(SIGTSTP-1)) // tty 发出的停止进程(tty stop)信号屏蔽位。
29
30 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
31 // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
32 #include <linux/tty.h> // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
33 #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
34 #include <asm/system.h> // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
35
36 #define L_FLAG(tty, f) ((tty->termios.c_lflag & f) // 取 termios 结构中的本地模式标志。
37 #define I_FLAG(tty, f) ((tty->termios.c_iflag & f) // 取 termios 结构中的输入模式标志。
38 #define O_FLAG(tty, f) ((tty->termios.c_oflag & f) // 取 termios 结构中的输出模式标志。
39
40 // 取 termios 结构中本地模式标志集中的一个标志位。
41 #define L_CANON(tty) L_FLAG((tty), ICANON) // 取本地模式标志集中规范(熟)模式标志位。
42 #define L_ISIG(tty) L_FLAG((tty), ISIG) // 取信号标志位。
43 #define L_ECHO(tty) L_FLAG((tty), ECHO) // 取回显字符标志位。
44 #define L_ECHOE(tty) L_FLAG((tty), ECHOE) // 规范模式时，取回显擦出标志位。
45 #define L_ECHOK(tty) L_FLAG((tty), ECHOK) // 规范模式时，取 KILL 擦除当前行标志位。
46 #define L_ECHOCTL(tty) L_FLAG((tty), ECHOCTL) // 取回显控制字符标志位。
47 #define L_ECHOKL(tty) L_FLAG((tty), ECHOKL) // 规范模式时，取 KILL 擦除行并回显标志位。

```



```

39 // 取 termios 结构中输入模式标志中的一个标志位。
40 #define I_UCLC(tty)      I_FLAG((tty), I_UCLC) // 取输入模式标志集中大写到低转换标志位。
41 #define I_NLCR(tty)      I_FLAG((tty), I_NLCR) // 取换行符 NL 转回车符 CR 标志位。
42 #define I_CRNL(tty)      I_FLAG((tty), I_CRNL) // 取回车符 CR 转换行符 NL 标志位。
43 #define I_NOCR(tty)      I_FLAG((tty), I_NOCR) // 取忽略回车符 CR 标志位。
44
45 // 取 termios 结构中输出模式标志中的一个标志位。
46 #define O_POST(tty)      O_FLAG((tty), O_POST) // 取输出模式标志集中执行输出处理标志。
47 #define O_NLCR(tty)      O_FLAG((tty), O_NLCR) // 取换行符 NL 转回车换行符 CR-NL 标志。
48 #define O_CRNL(tty)      O_FLAG((tty), O_CRNL) // 取回车符 CR 转换行符 NL 标志。
49 #define O_NLRET(tty)     O_FLAG((tty), O_NLRET) // 取换行符 NL 执行回车功能的标志。
50 #define O_LCUC(tty)      O_FLAG((tty), O_LCUC) // 取小写转大写字符标志。
51
52 // tty 数据结构的 tty_table 数组。其中包含三个初始化项数据，分别对应控制台、串口终端 1 和
53 // 串口终端 2 的初始化数据。
54 struct tty_struct tty_table[] = {
55     {
56         {ICRNL, /* change incoming CR to NL */ /* 将输入的 CR 转换为 NL */
57         OPOST | ONLCR, /* change outgoing NL to CRNL */ /* 将输出的 NL 转 CRNL */
58         0, // 控制模式标志初始化为 0。
59         ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, // 本地模式标志。
60         0, /* console termio */ // 控制台 termio。
61         INIT_C_CC}, // 控制字符数组。
62         0, /* initial pgrp */ // 所属初始进程组。
63         0, /* initial stopped */ // 初始停止标志。
64         con_write, // tty 写函数指针。
65         {0, 0, 0, 0, ""}, /* console read-queue */ // tty 控制台读队列。
66         {0, 0, 0, 0, ""}, /* console write-queue */ // tty 控制台写队列。
67         {0, 0, 0, 0, ""} /* console secondary queue */ // tty 控制台辅助(第二)队列。
68     }, {
69         {0, /* no translation */ // 输入模式标志。0, 无须转换。
70         0, /* no translation */ // 输出模式标志。0, 无须转换。
71         B2400 | CS8, // 控制模式标志。波特率 2400bps, 8 位数据位。
72         0, // 本地模式标志 0。
73         0, // 行规程 0。
74         INIT_C_CC}, // 控制字符数组。
75         0, // 所属初始进程组。
76         0, // 初始停止标志。
77         rs_write, // 串口 1 tty 写函数指针。
78         {0x3f8, 0, 0, 0, ""}, /* rs 1 */ // 串行终端 1 读缓冲队列。
79         {0x3f8, 0, 0, 0, ""}, // 串行终端 1 写缓冲队列。
80         {0, 0, 0, 0, ""} // 串行终端 1 辅助缓冲队列。
81     }, {
82         {0, /* no translation */ // 输入模式标志。0, 无须转换。
83         0, /* no translation */ // 输出模式标志。0, 无须转换。
84         B2400 | CS8, // 控制模式标志。波特率 2400bps, 8 位数据位。
85         0, // 本地模式标志 0。
86         0, // 行规程 0。
87         INIT_C_CC}, // 控制字符数组。
88         0, // 所属初始进程组。
89         0, // 初始停止标志。
90         rs_write, // 串口 2 tty 写函数指针。

```



```

88         {0x2f8, 0, 0, 0, ""},          /* rs 2 */ // 串行终端 2 读缓冲队列。
89         {0x2f8, 0, 0, 0, ""},          // 串行终端 2 写缓冲队列。
90         {0, 0, 0, 0, ""}               // 串行终端 2 辅助缓冲队列。
91     }
92 };
93
94 /*
95  * these are the tables used by the machine code handlers.
96  * you can implement pseudo-tty's or something by changing
97  * them. Currently not done.
98  */
99 /*
100  * 下面是汇编程序使用的缓冲队列地址表。通过修改你可以实现
101  * 伪 tty 终端或其它终端类型。目前还没有这样做。
102  */
103 // tty 缓冲队列地址表。rs_io.s 汇编程序使用，用于取得读写缓冲队列地址。
104 struct tty_queue * table_list[]={
105     &tty_table[0].read_q, &tty_table[0].write_q, // 控制台终端读、写缓冲队列地址。
106     &tty_table[1].read_q, &tty_table[1].write_q, // 串行口 1 终端读、写缓冲队列地址。
107     &tty_table[2].read_q, &tty_table[2].write_q // 串行口 2 终端读、写缓冲队列地址。
108 };
109
110 // tty 终端初始化函数。
111 // 初始化串口终端和控制台终端。
112 void tty_init(void)
113 {
114     rs_init(); // 初始化串行中断程序和串行接口 1 和 2。(serial.c, 37)
115     con_init(); // 初始化控制台终端。(console.c, 617)
116 }
117
118 // tty 键盘中断 (^C) 字符处理函数。
119 // 向 tty 结构中指明的 (前台) 进程组中所有的进程发送指定的信号 mask, 通常该信号是 SIGINT。
120 // 参数: tty - 相应 tty 终端结构指针; mask - 信号屏蔽位。
121 void tty_intr(struct tty_struct * tty, int mask)
122 {
123     int i;
124
125     // 如果 tty 所属进程组号小于等于 0, 则退出。
126     // 当 pgrp=0 时, 表明进程是初始进程 init, 它没有控制终端, 因此不应该发出中断字符。
127     if (tty->pgrp <= 0)
128         return;
129     // 扫描任务数组, 向 tty 指明的进程组 (前台进程组) 中所有进程发送指定的信号。
130     for (i=0; i<NR_TASKS; i++)
131     // 如果该项任务指针不为空, 并且其组号等于 tty 组号, 则设置 (发送) 该任务指定的信号 mask。
132         if (task[i] && task[i]->pgrp==tty->pgrp)
133             task[i]->signal |= mask;
134 }
135
136 // tty 队列缓冲区空则让进程进入可中断的睡眠状态。
137 // 参数: queue - 指定队列的指针。
138 // 进程在取队列缓冲区中字符时调用此函数。
139 static void sleep_if_empty(struct tty_queue * queue)
140 {

```

```

124     cli(); // 关中断。
// 若当前进程没有信号要处理并且指定的队列缓冲区空，则让进程进入可中断睡眠状态，并让
// 队列的进程等待指针指向该进程。
125     while (!current->signal && EMPTY(*queue))
126         interruptible_sleep_on(&queue->proc_list);
127     sti(); // 开中断。
128 }
129
//// 若队列缓冲区满则让进程进入可中断的睡眠状态。
// 参数: queue - 指定队列的指针。
// 进程在往队列缓冲区中写入时调用此函数。
130 static void sleep_if_full(struct tty_queue * queue)
131 {
// 若队列缓冲区不满，则返回退出。
132     if (!FULL(*queue))
133         return;
134     cli(); // 关中断。
// 如果进程没有信号需要处理并且队列缓冲区内空闲剩余区长度<128，则让进程进入可中断睡眠状态，
// 并让该队列的进程等待指针指向该进程。
135     while (!current->signal && LEFT(*queue)<128)
136         interruptible_sleep_on(&queue->proc_list);
137     sti(); // 开中断。
138 }
139
//// 等待按键。
// 如果控制台的读队列缓冲区空则让进程进入可中断的睡眠状态。
140 void wait_for_keypress(void)
141 {
142     sleep_if_empty(&tty_table[0].secondary);
143 }
144
//// 复制成规范模式字符序列。
// 将指定 tty 终端队列缓冲区中的字符复制成规范(熟)模式字符并存放在辅助队列(规范模式队列)中。
// 参数: tty - 指定终端的 tty 结构。
145 void copy_to_cooked(struct tty_struct * tty)
146 {
147     signed char c;
148
// 如果 tty 的读队列缓冲区不空并且辅助队列缓冲区为空，则循环执行下列代码。
149     while (!EMPTY(tty->read_q) && !FULL(tty->secondary)) {
// 从读队列尾处取一字符到 c，并前移尾指针。
150         GETCH(tty->read_q, c);
// 下面对输入字符，利用输入模式标志集进行处理。
// 如果该字符是回车符 CR(13)，则：若回车转换行标志 CRNL 置位则将该字符转换为换行符 NL(10)；
// 否则若忽略回车标志 NOCR 置位，则忽略该字符，继续处理其它字符。
151         if (c==13)
152             if (I_CRNL(tty))
153                 c=10;
154             else if (I_NOCR(tty))
155                 continue;
156             else ;
// 如果该字符是换行符 NL(10)并且换行转回车标志 NLCR 置位，则将其转换为回车符 CR(13)。
157         else if (c==10 && I_NLCR(tty))

```

```

158             c=13;
// 如果大写转小写标志 UCLC 置位，则将该字符转换为小写字符。
159             if (I_UCLC(tty))
160                 c=tolower(c);
// 如果本地模式标志集中规范（熟）模式标志 CANON 置位，则进行以下处理。
161             if (L_CANON(tty)) {
// 如果该字符是键盘终止控制字符 KILL(^U)，则进行删除输入行上所有字符的处理。
162                 if (c==KILL_CHAR(tty)) {
163                     /* deal with killing the input line */ /* 删除输入行处理 */
// 如果 tty 辅助队列不空，或者辅助队列中最后一个字符是换行 NL(10)，或者该字符是文件结束字符
// (^D)，则循环执行下列代码。
164                     while(!EMPTY(tty->secondary) ||
165                           (c=LAST(tty->secondary))==10 ||
166                           c==EOF_CHAR(tty))) {
// 如果本地回显标志 ECHO 置位，那么：若字符是控制字符(值<32)，则往 tty 的写队列中放入擦除
// 字符 ERASE。再放入一个擦除字符 ERASE，并且调用该 tty 的写函数，该写函数就会把写队列中的
// 字符输出到终端的屏幕上。另外，因为控制字符在放入写队列时是用两个字符表示的（例如^V），
// 因此需要特别对控制字符多放入一个 ERASE。
167                     if (L_ECHO(tty)) {
168                         if (c<32)
169                             PUTCH(127,tty->write_q);
170                             PUTCH(127,tty->write_q);
171                             tty->write(tty);
172                     }
// 将 tty 辅助队列头指针后退 1 字节。
173                     DEC(tty->secondary.head);
174                 }
175                 continue;    // 继续读取并处理其它字符。
176             }
// 如果该字符是删除控制字符 ERASE(^H)，那么：
177             if (c==ERASE_CHAR(tty)) {
// 若 tty 的辅助队列为空，或者其最后一个字符是换行符 NL(10)，或者是文件结束符，则继续处理
// 其它字符。
178                 if (EMPTY(tty->secondary) ||
179                     (c=LAST(tty->secondary))==10 ||
180                     c==EOF_CHAR(tty))
181                     continue;
// 如果本地回显标志 ECHO 置位，那么：若字符是控制字符(值<32)，则往 tty 的写队列中放入擦除
// 字符 ERASE。再放入一个擦除字符 ERASE，并且调用该 tty 的写函数。
182                 if (L_ECHO(tty)) {
183                     if (c<32)
184                         PUTCH(127,tty->write_q);
185                         PUTCH(127,tty->write_q);
186                         tty->write(tty);
187                 }
// 将 tty 辅助队列头指针后退 1 字节，继续处理其它字符。
188                 DEC(tty->secondary.head);
189                 continue;
190             }
//如果该字符是停止字符(^S)，则置 tty 停止标志，继续处理其它字符。
191             if (c==STOP_CHAR(tty)) {
192                 tty->stopped=1;
193                 continue;

```

```

194     }
    // 如果该字符是开始字符(^Q), 则复位 tty 停止标志, 继续处理其它字符。
195     if (c==START_CHAR(tty)) {
196         tty->stopped=0;
197         continue;
198     }
199 }
// 若输入模式标志集中 ISIG 标志置位, 表示终端键盘可以产生信号, 则在收到 INTR、QUIT、SUSP
// 或 DSUSP 字符时, 需要为进程产生相应的信号。
200     if (L_ISIG(tty)) {
    // 如果该字符是键盘中断符(^C), 则向当前进程发送键盘中断信号, 并继续处理下一字符。
201         if (c==INTR_CHAR(tty)) {
202             tty_intr(tty, INTRMASK);
203             continue;
204         }
    // 如果该字符是退出符(^_), 则向当前进程发送键盘退出信号, 并继续处理下一字符。
205         if (c==QUIT_CHAR(tty)) {
206             tty_intr(tty, QUITMASK);
207             continue;
208         }
209     }
    // 如果该字符是换行符 NL(10), 或者是文件结束符 EOF(4, ^D), 表示一行字符已处理完, 则把辅助
    // 缓冲队列中含有字符行数值增 1。在 tty_read() 中若取走一行字符, 就会将其值减 1, 见 264 行。
210     if (c==10 || c==EOF_CHAR(tty))
211         tty->secondary.data++;
    // 如果本地模式标志集中回显标志 ECHO 置位, 那么, 如果字符是换行符 NL(10), 则将换行符 NL(10)
    // 和回车符 CR(13)放入 tty 写队列缓冲区中; 如果字符是控制字符(字符值<32)并且回显控制字符标志
    // ECHOCTL 置位, 则将字符'^'和字符 c+64 放入 tty 写队列中(也即会显示^C、^H等); 否则将该字符
    // 直接放入 tty 写缓冲队列中。最后调用该 tty 的写操作函数。
212     if (L_ECHO(tty)) {
213         if (c==10) {
214             PUTCH(10, tty->write_q);
215             PUTCH(13, tty->write_q);
216         } else if (c<32) {
217             if (L_ECHOCTL(tty)) {
218                 PUTCH('^', tty->write_q);
219                 PUTCH(c+64, tty->write_q);
220             }
221         } else
222             PUTCH(c, tty->write_q);
223         tty->write(tty);
224     }
    // 将该字符放入辅助队列中。
225     PUTCH(c, tty->secondary);
226 }
    // 唤醒等待该辅助缓冲队列的进程(如果有的话)。
227     wake_up(&tty->secondary.proc_list);
228 }
229
    //// tty 读函数, 从终端辅助缓冲队列中读取指定数量的字符, 放到用户指定的缓冲区中。
    // 参数: channel - 子设备号; buf - 用户缓冲区指针; nr - 欲读字节数。
    // 返回已读字节数。
230 int tty_read(unsigned channel, char * buf, int nr)

```

```

231 {
232     struct tty\_struct * tty;
233     char c, * b=buf;
234     int minimum, time, flag=0;
235     long oldalarm;
236
237     // 本版本 linux 内核的终端只有 3 个子设备，分别是控制台 (0)、串口终端 1 (1) 和串口终端 2 (2)。
238     // 所以任何大于 2 的子设备号都是非法的。读的字节数当然也不能小于 0 的。
239     if (channel>2 || nr<0) return -1;
240     // tty 指针指向子设备号对应 ttb_table 表中的 tty 结构。
241     tty = &tty\_table[channel];
242     // 下面首先保存进程原定时值，然后根据控制字符 VTIME 和 VMIN 设置读字符操作的超时定时值。
243     // 在非规范模式下，这两个值是超时定时值。MIN 表示为了满足读操作，需要读取的最少字符数。
244     // TIME 是一个十分之一秒计数的计时值。
245     // 首先保存进程当前的(报警)定时值(滴答数)。
246     oldalarm = current->alarm;
247     // 并设置读操作超时定时值 time 和需要最少读取的字符个数 minimum。
248     time = 10L*tty->termios.c_cc[VTIME];
249     minimum = tty->termios.c_cc[VMIN];
250     // 如果设置了读超时定时值 time 但没有设置最少读取个数 minimum，那么在读到至少一个字符或者
251     // 定时超时后读操作将立刻返回。所以这里置 minimum=1。
252     if (time && !minimum) {
253         minimum=1;
254         // 如果进程原定时值是 0 或者 time+当前系统时间值小于进程原定时值的话，则置重新设置进程定时
255         // 值为 time+当前系统时间，并置 flag 标志。
256         if (flag=(!oldalarm || time+jiffies<oldalarm))
257             current->alarm = time+jiffies;
258     }
259     // 如果设置的最少读取字符数>欲读的字符数，则令其等于此次欲读取的字符数。
260     if (minimum>nr)
261         minimum=nr;
262     // 当欲读的字节数>0，则循环执行以下操作。
263     while (nr>0) {
264         // 如果 flag 不为 0 (也即进程原定时值是 0 或者 time+当前系统时间值小于进程原定时值) 并且进程此时
265         // 以收到定时信号 SIGALRM，表明这里设置的定时时间已到，则复位进程的定时信号 SIGALRM，并中断
266         // 循环。
267         if (flag && (current->signal & ALRMASK)) {
268             current->signal &= ~ALRMASK;
269             break;
270         }
271         // 如果 flag 没有置位，或者已置位但当前进程有其它信号要处理，则退出，返回 0。
272         if (current->signal)
273             break;
274         // 如果辅助缓冲队列(规范模式队列)为空，或者设置了规范模式标志并且辅助队列中字符数为 0 以及
275         // 辅助模式缓冲队列空闲空间>20，则进入可中断睡眠状态，返回后继续处理。
276         if (EMPTY(tty->secondary) || (L\_CANON(tty) &&
277             !tty->secondary.data && LEFT(tty->secondary)>20)) {
278             sleep\_if\_empty(&tty->secondary);
279             continue;
280         }
281         // 执行以下取字符操作，需读字符数 nr 依次递减，直到 nr=0 或者辅助缓冲队列为空。
282         do {
283             // 取辅助缓冲队列字符 c，并且缓冲队列 secondary->tail 指针向右移动一个字符位置 (tail++)。

```

```

262         GETCH(tty->secondary, c);
// 如果该字符是文件结束符(^D)或者是换行符 NL(10), 则把辅助缓冲队列中含有字符行数减 1。
263         if (c==EOF_CHAR(tty) || c==10)
264             tty->secondary.data--;
// 如果该字符是文件结束符(^D)并且规范模式标志置位, 则返回已读字符数, 并退出。
265         if (c==EOF_CHAR(tty) && L_CANON(tty))
266             return (b-buf);
// 否则说明是原始模式(非规范模式)操作, 于是将该字符直接放入用户数据段缓冲区 buf 中, 并把
// 欲读字符数减 1。此时如果欲读字符数已为 0, 则中断循环。
267         else {
268             put_fs_byte(c, b++);
269             if (!--nr)
270                 break;
271         }
272     } while (nr>0 && !EMPTY(tty->secondary));
// 如果超时定时值 time 不为 0 并且规范模式标志没有置位(非规范模式), 那么:
273     if (time && !L_CANON(tty))
// 如果进程原定时值是 0 或者 time+当前系统时间值小于进程原定时值的话, 则置重新设置进程定时值
// 为 time+当前系统时间, 并置 flag 标志, 为读取下一个字符做好定时准备。否则表明进程原定时时间
// 要比等待一个字符被读取的定时时间要小, 可能没有等到字符到来进程原定时时间就到了。因此,
// 此时这里需要恢复进程的原定时值(oldalarm)。
274         if (flag!=(oldalarm || time+jiffies<oldalarm))
275             current->alarm = time+jiffies;
276         else
277             current->alarm = oldalarm;
// 如果规范模式标志置位, 那么若已读到起码一个字符则中断循环。否则若已读取数大于或等于最少要
// 求读取的字符数, 则也中断循环。
278         if (L_CANON(tty)) {
279             if (b-buf)
280                 break;
281         } else if (b-buf >= minimum)
282             break;
283     }
// 读取 tty 字符循环操作结束, 让进程的定时值恢复值。
284     current->alarm = oldalarm;
// 如果进程有信号并且没有读取到任何字符, 则返回出错号(被中断)。
285     if (current->signal && !(b-buf))
286         return -EINTR;
287     return (b-buf); // 返回已读取的字符数。
288 }
289
//// tty 写函数。把用户缓冲区中的字符写入 tty 的写队列中。
// 参数: channel - 子设备号; buf - 缓冲区指针; nr - 写字节数。
// 返回已写字节数。
290 int tty_write(unsigned channel, char * buf, int nr)
291 {
292     static cr_flag=0;
293     struct tty_struct * tty;
294     char c, *b=buf;
295
// 本版本 linux 内核的终端只有 3 个子设备, 分别是控制台(0)、串口终端 1(1)和串口终端 2(2)。
// 所以任何大于 2 的子设备号都是非法的。写的字节数当然也不能小于 0 的。
296     if (channel>2 || nr<0) return -1;

```

```

// tty 指针指向子设备号对应 ttb_table 表中的 tty 结构。与第 238 行语句的作用相同。
297     tty = channel + tty\_table;
// 字符设备是一个一个字符进行处理的，所以这里对于 nr 大于 0 时对每个字符进行循环处理。
298     while (nr>0) {
// 如果此时 tty 的写队列已满，则当前进程进入可中断的睡眠状态。
299         sleep\_if\_full(&tty->write_q);
// 如果当前进程有信号要处理，则退出，返回 0。
300         if (current->signal)
301             break;
// 当要写的字节数>0 并且 tty 的写队列不满时，循环执行以下操作。
302         while (nr>0 && !FULL(tty->write_q)) {
// 从用户数据段内存中取一字节 c。
303             c=get\_fs\_byte(b);
// 如果终端输出模式标志集中的执行输出处理标志 OPOST 置位，则执行下列输出时处理过程。
304             if (O\_POST(tty)) {
// 如果该字符是回车符 '\r' (CR, 13) 并且回车符转换行符标志 OCRNL 置位，则将该字符换成换行符
// '\n' (NL, 10)；否则如果该字符是换行符 '\n' (NL, 10) 并且换行转回车功能标志 ONLRET 置位的话，
// 则将该字符换成回车符 '\r' (CR, 13)。
305                 if (c=='\r' && O\_CRNL(tty))
306                     c='\n';
307                 else if (c=='\n' && O\_NLRET(tty))
308                     c='\r';
// 如果该字符是换行符 '\n' 并且回车标志 cr_flag 没有置位，换行转回车-换行标志 ONLCR 置位的话，
// 则将 cr_flag 置位，并将一回车符放入写队列中。然后继续处理下一个字符。
309                 if (c=='\n' && !cr_flag && O\_NLCR(tty)) {
310                     cr_flag = 1;
311                     PUTCH(13, tty->write_q);
312                     continue;
313                 }
// 如果小写转大写标志 OLCUC 置位的话，就将该字符转成大写字符。
314                 if (O\_LCUC(tty))
315                     c=toupper(c);
316             }
// 用户数据缓冲指针 b 前进 1 字节；欲写字节数减 1 字节；复位 cr_flag 标志，并将该字节放入 tty
// 写队列中。
317             b++; nr--;
318             cr_flag = 0;
319             PUTCH(c, tty->write_q);
320         }
// 若字节全部写完，或者写队列已满，则程序执行到这里。调用对应 tty 的写函数，若还有字节要写，
// 则等待写队列不满，所以调用调度程序，先去执行其它任务。
321         tty->write(tty);
322         if (nr>0)
323             schedule();
324     }
325     return (b-buf); // 返回写入的字节数。
326 }
327
328 /*
329  * Jeh, sometimes I really like the 386.
330  * This routine is called from an interrupt,
331  * and there should be absolutely no problem
332  * with sleeping even in an interrupt (I hope).

```



```

333  * Of course, if somebody proves me wrong, I'll
334  * hate intel for all time :-). We'll have to
335  * be careful and see to reinstating the interrupt
336  * chips before calling this, though.
337  *
338  * I don't think we sleep here under normal circumstances
339  * anyway, which is good, as the task sleeping might be
340  * totally innocent.
341  */
/*
* 呵，有时我是真得很喜欢 386。该子程序是从一个中断处理程序中调用的，即使在
* 中断处理程序中睡眠也应该绝对没有问题(我希望如此)。当然，如果有人证明我是
* 错的，那么我将憎恨 intel 一辈子☺。但是我们必须小心，在调用该子程序之前需
* 要恢复中断。
*
* 我不认为在通常环境下会处在这里睡眠，这样很好，因为任务睡眠是完全任意的。
*/
///// tty 中断处理调用函数 - 执行 tty 中断处理。
// 参数: tty - 指定的 tty 终端号 (0, 1 或 2)。
// 将指定 tty 终端队列缓冲区中的字符复制成规范(熟)模式字符并存放在辅助队列(规范模式队列)中。
// 在串口读字符中断(rs_io.s, 109)和键盘中断(kerboard.S, 69)中调用。
342 void do_tty_interrupt(int tty)
343 {
344     copy_to_cooked(tty_table+tty);
345 }
346
///// 字符设备初始化函数。空，为以后扩展做准备。
347 void chr_dev_init(void)
348 {
349 }
350

```

7.8.3 其它信息

7.8.3.1 控制字符 VTIME、VMIN

在非规范模式下，这两个值是超时定时值和最小读取字符个数。MIN 表示为了满足读操作，需要读取的最少字符数。TIME 是一个十分之一秒计数的计时值。当这两个都设置的话，读操作将等待，直到至少读到一个字符，然后在以读取 MIN 个字符或者时间 TIME 在读取最后一个字符后超时。如果仅设置了 MIN，那么在读取 MIN 个字符之前读操作将不返回。如果仅设置了 TIME，那么在读到至少一个字符或者定时超时后读操作将立刻返回。如果两个都没有设置，则读操作将立刻返回，仅给出目前已读的字节数。详细说明参见 `termios.h` 文件。

7.9 tty_ioctl.c 程序

7.9.1 功能描述

本文件用于字符设备的控制操作，实现了函数（系统调用）`tty_ioctl()`。程序通过使用该函数可以修改指定终端 `termios` 结构中的设置标志等信息。

7.9.2 代码注释

程序 7-7 linux/kernel/chr_drv/tty_ioctl.c

```

1  /*
2   *  linux/kernel/chr_drv/tty_ioctl.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
8  #include <termios.h>       // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
9
10 #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
11 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
13
14 #include <asm/io.h>         // io 头文件。定义硬件端口输入/输出宏汇编语句。
15 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
16 #include <asm/system.h>     // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
17
    // 这是波特率因子数组（或称为除数数组）。波特率与波特率因子的对应关系参见列表后的说明。
18 static unsigned short quotient[] = {
19     0, 2304, 1536, 1047, 857,
20     768, 576, 384, 192, 96,
21     64, 48, 24, 12, 6, 3
22 };
23
    // 修改传输速率。
    // 参数：tty - 终端对应的 tty 数据结构。
    // 在除数锁存标志 DLAB(线路控制寄存器位 7) 置位情况下，通过端口 0x3f8 和 0x3f9 向 UART 分别写入
    // 波特率因子低字节和高字节。
24 static void change_speed(struct tty_struct * tty)
25 {
26     unsigned short port, quot;
27
    // 对于串口终端，其 tty 结构的读缓冲队列 data 字段存放的是串行端口号(0x3f8 或 0x2f8)。
28     if (!(port = tty->read_q.data))
29         return;
    // 从 tty 的 termios 结构控制模式标志集中取得设置的波特率索引号，据此从波特率因子数组中取得
    // 对应的波特率因子值。CBAUD 是控制模式标志集中波特率位屏蔽码。
30     quot = quotient[tty->termios.c_cflag & CBAUD];
31     cli();                    // 关中断。
32     outb_p(0x80, port+3);     /* set DLAB */ // 首先设置除数锁定标志 DLAB。
33     outb_p(quot & 0xff, port); /* LS of divisor */ // 输出因子低字节。
34     outb_p(quot >> 8, port+1); /* MS of divisor */ // 输出因子高字节。
35     outb(0x03, port+3);       /* reset DLAB */ // 复位 DLAB。
36     sti();                    // 开中断。
37 }
38
    // 刷新 tty 缓冲队列。
    // 参数：queue - 指定的缓冲队列指针。

```

```

// 令缓冲队列的头指针等于尾指针，从而达到清空缓冲区(零字符)的目的。
39 static void flush(struct tty_queue * queue)
40 {
41     cli();
42     queue->head = queue->tail;
43     sti();
44 }
45
//// 等待字符发送出去。
46 static void wait_until_sent(struct tty_struct * tty)
47 {
48     /* do nothing - not implemented */ /* 什么都没做 - 还未实现 */
49 }
50
//// 发送 BREAK 控制符。
51 static void send_break(struct tty_struct * tty)
52 {
53     /* do nothing - not implemented */ /* 什么都没做 - 还未实现 */
54 }
55
//// 取终端 termios 结构信息。
// 参数: tty - 指定终端的 tty 结构指针; termios - 用户数据区 termios 结构缓冲区指针。
// 返回 0 。
56 static int get_termios(struct tty_struct * tty, struct termios * termios)
57 {
58     int i;
59
// 首先验证一下用户的缓冲区指针所指内存区是否足够，如不够则分配内存。
60     verify_area(termios, sizeof (*termios));
// 复制指定 tty 结构中的 termios 结构信息到用户 termios 结构缓冲区。
61     for (i=0 ; i< (sizeof (*termios)) ; i++)
62         put_fs_byte( ((char *)&tty->termios)[i] , i+(char *)termios );
63     return 0;
64 }
65
//// 设置终端 termios 结构信息。
// 参数: tty - 指定终端的 tty 结构指针; termios - 用户数据区 termios 结构指针。
// 返回 0 。
66 static int set_termios(struct tty_struct * tty, struct termios * termios)
67 {
68     int i;
69
// 首先复制用户数据区中 termios 结构信息到指定 tty 结构中。
70     for (i=0 ; i< (sizeof (*termios)) ; i++)
71         ((char *)&tty->termios)[i]=get_fs_byte(i+(char *)termios);
// 用户有可能已修改了 tty 的串行口传输波特率，所以根据 termios 结构中的控制模式标志 c_cflag
// 修改串行芯片 UART 的传输波特率。
72     change_speed(tty);
73     return 0;
74 }
75

//// 读取 termio 结构中的信息。
// 参数: tty - 指定终端的 tty 结构指针; termio - 用户数据区 termio 结构缓冲区指针。

```

```

// 返回 0。
86 static int get_termio(struct tty_struct * tty, struct termio * termio)
87 {
88     int i;
89     struct termio tmp_termio;
90
91 // 首先验证一下用户的缓冲区指针所指内存区是否足够，如不够则分配内存。
92 verify_area(termio, sizeof (*termio));
93 // 将 termios 结构的信息复制到 termio 结构中。目的是为了其中模式标志集的类型进行转换，也即
94 // 从 termios 的长整数类型转换为 termio 的短整数类型。
95 tmp_termio.c_iflag = tty->termios.c_iflag;
96 tmp_termio.c_oflag = tty->termios.c_oflag;
97 tmp_termio.c_cflag = tty->termios.c_cflag;
98 tmp_termio.c_lflag = tty->termios.c_lflag;
99 // 两种结构的 c_line 和 c_cc[] 字段是完全相同的。
100 tmp_termio.c_line = tty->termios.c_line;
101 for(i=0 ; i < NCC ; i++)
102     tmp_termio.c_cc[i] = tty->termios.c_cc[i];
103 // 最后复制指定 tty 结构中的 termio 结构信息到用户 termio 结构缓冲区。
104 for (i=0 ; i < (sizeof (*termio)) ; i++)
105     put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio );
106 return 0;
107 }
108
109 /*
110 * This only works as the 386 is low-byt-first
111 */
112 /*
113 * 下面的 termio 设置函数仅在 386 低字节在前的方式下可用。
114 */
115 // 设置终端 termio 结构信息。
116 // 参数: tty - 指定终端的 tty 结构指针; termio - 用户数据区 termio 结构指针。
117 // 将用户缓冲区 termio 的信息复制到终端的 termios 结构中。返回 0 。
118 static int set_termio(struct tty_struct * tty, struct termio * termio)
119 {
120     int i;
121     struct termio tmp_termio;
122
123 // 首先复制用户数据区中 termio 结构信息到临时 termio 结构中。
124 for (i=0 ; i < (sizeof (*termio)) ; i++)
125     ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
126 // 再将 termio 结构的信息复制到 tty 的 termios 结构中。目的是为了其中模式标志集的类型进行转换，
127 // 也即从 termio 的短整数类型转换成 termios 的长整数类型。
128 *(unsigned short *)&tty->termios.c_iflag = tmp_termio.c_iflag;
129 *(unsigned short *)&tty->termios.c_oflag = tmp_termio.c_oflag;
130 *(unsigned short *)&tty->termios.c_cflag = tmp_termio.c_cflag;
131 *(unsigned short *)&tty->termios.c_lflag = tmp_termio.c_lflag;
132 // 两种结构的 c_line 和 c_cc[] 字段是完全相同的。
133 tty->termios.c_line = tmp_termio.c_line;
134 for(i=0 ; i < NCC ; i++)
135     tty->termios.c_cc[i] = tmp_termio.c_cc[i];
136 // 用户可能已修改了 tty 的串行口传输波特率，所以根据 termios 结构中的控制模式标志集 c_cflag
137 // 修改串行芯片 UART 的传输波特率。

```

```

111     change_speed(tty);
112     return 0;
113 }
114
115 // tty 终端设备的 ioctl 函数。
116 // 参数: dev - 设备号; cmd - ioctl 命令; arg - 操作参数指针。
117 int tty_ioctl(int dev, int cmd, int arg)
118 {
119     struct tty_struct * tty;
120     // 首先取 tty 的子设备号。如果主设备号是 5(tty 终端), 则进程的 tty 字段即是子设备号; 如果进程
121     // 的 tty 子设备号是负数, 表明该进程没有控制终端, 也即不能发出该 ioctl 调用, 出错死机。
122     if (MAJOR(dev) == 5) {
123         dev=current->tty;
124         if (dev<0)
125             panic("tty_ioctl: dev<0");
126         // 否则直接从设备号中取出子设备号。
127     } else
128         dev=MINOR(dev);
129     // 子设备号可以是 0(控制台终端)、1(串口 1 终端)、2(串口 2 终端)。
130     // 让 tty 指向对应子设备号的 tty 结构。
131     tty = dev + tty_table;
132     // 根据 tty 的 ioctl 命令进行分别处理。
133     switch (cmd) {
134         case TCGETS:
135             //取相应终端 termios 结构中的信息。
136             return get_termios(tty, (struct termios *) arg);
137         case TCSETSF:
138             // 在设置 termios 的信息之前, 需要先等待输出队列中所有数据处理完, 并且刷新(清空)输入队列。
139             // 再设置。
140             flush(&tty->read_q); /* fallthrough */
141         case TCSETSW:
142             // 在设置终端 termios 的信息之前, 需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
143             // 会影响输出的情况, 就需要使用这种形式。
144             wait_until_sent(tty); /* fallthrough */
145         case TCSETS:
146             // 设置相应终端 termios 结构中的信息。
147             return set_termios(tty, (struct termios *) arg);
148         case TCGETA:
149             // 取相应终端 termio 结构中的信息。
150             return get_termio(tty, (struct termio *) arg);
151         case TCSETAF:
152             // 在设置 termio 的信息之前, 需要先等待输出队列中所有数据处理完, 并且刷新(清空)输入队列。
153             // 再设置。
154             flush(&tty->read_q); /* fallthrough */
155         case TCSETAW:
156             // 在设置终端 termio 的信息之前, 需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
157             // 会影响输出的情况, 就需要使用这种形式。
158             wait_until_sent(tty); /* fallthrough */ /* 继续执行 */
159         case TCSETA:
160             // 设置相应终端 termio 结构中的信息。
161             return set_termio(tty, (struct termio *) arg);
162         case TCSBRK:
163             // 等待输出队列处理完毕(空), 如果参数值是 0, 则发送一个 break。

```

```

143         if (!arg) {
144             wait\_until\_sent(tty);
145             send\_break(tty);
146         }
147         return 0;
148     case TCXONC:
149         // 开始/停止控制。如果参数值是 0，则挂起输出；如果是 1，则重新开启挂起的输出；如果是 2，则挂
150         // 起
151         // 输入；如果是 3，则重新开启挂起的输入。
152         return -EINVAL; /* not implemented */ /* 未实现 */
153     case TCFLSH:
154         //刷新已写输出但还没发送或已收但还没有读数据。如果参数是 0，则刷新(清空)输入队列；如果是 1，
155         // 则刷新输出队列；如果是 2，则刷新输入和输出队列。
156         if (arg==0)
157             flush(&tty->read_q);
158         else if (arg==1)
159             flush(&tty->write_q);
160         else if (arg==2) {
161             flush(&tty->read_q);
162             flush(&tty->write_q);
163         } else
164             return -EINVAL;
165         return 0;
166     case TIOCEXCL:
167         // 设置终端串行线路专用模式。
168         return -EINVAL; /* not implemented */ /* 未实现 */
169     case TIOCNXCL:
170         // 复位终端串行线路专用模式。
171         return -EINVAL; /* not implemented */ /* 未实现 */
172     case TIOCSCTTY:
173         // 设置 tty 为控制终端。(TIOCNOTTY - 禁止 tty 为控制终端)。
174         return -EINVAL; /* set controlling term NI */ /* 设置控制终端 NI */
175     case TIOCGPGRP:
176         // NI - Not Implemented。
177         // 读取指定终端设备进程的组 id。首先验证用户缓冲区长度，然后复制 tty 的 pgrp 字段到用户缓冲区。
178         verify\_area((void *) arg, 4);
179         put\_fs\_long(tty->pgrp, (unsigned long *) arg);
180         return 0;
181     case TIOCSGRP:
182         // 设置指定终端设备进程的组 id。
183         tty->pgrp=get\_fs\_long((unsigned long *) arg);
184         return 0;
185     case TIOCOUTQ:
186         // 返回输出队列中还未送出的字符数。首先验证用户缓冲区长度，然后复制队列中字符数给用户。
187         verify\_area((void *) arg, 4);
188         put\_fs\_long(CHARS(tty->write_q), (unsigned long *) arg);
189         return 0;
190     case TIOCINQ:
191         // 返回输入队列中还未读取的字符数。首先验证用户缓冲区长度，然后复制队列中字符数给用户。
192         verify\_area((void *) arg, 4);
193         put\_fs\_long(CHARS(tty->secondary),
194             (unsigned long *) arg);
195         return 0;
196     case TIOCSTI:

```

```

// 模拟终端输入。该命令以一个指向字符的指针作为参数，并假装该字符是在终端上键入的。用户必须
// 在该控制终端上具有超级用户权限或具有读许可权限。
184         return -EINVAL; /* not implemented */ /* 未实现 */
185     case TIOCGWINSZ:
// 读取终端设备窗口大小信息（参见 termios.h 中的 winsize 结构）。
186         return -EINVAL; /* not implemented */ /* 未实现 */
187     case TIOCSWINSZ:
// 设置终端设备窗口大小信息（参见 winsize 结构）。
188         return -EINVAL; /* not implemented */ /* 未实现 */
189     case TIOCMGET:
// 返回 modem 状态控制引线的当前状态比特位标志集（参见 termios.h 中 185-196 行）。
190         return -EINVAL; /* not implemented */ /* 未实现 */
191     case TIOCMBIS:
// 设置单个 modem 状态控制引线的状态(true 或 false)。
192         return -EINVAL; /* not implemented */ /* 未实现 */
193     case TIOCMBIC:
// 复位单个 modem 状态控制引线的状态。
194         return -EINVAL; /* not implemented */ /* 未实现 */
195     case TIOCMSET:
// 设置 modem 状态引线的状态。如果某一比特位置位，则 modem 对应的状态引线将置为有效。
196         return -EINVAL; /* not implemented */ /* 未实现 */
197     case TIOCGSOFTCAR:
// 读取软件载波检测标志(1 - 开启; 0 - 关闭)。
198         return -EINVAL; /* not implemented */ /* 未实现 */
199     case TIOCSSOFTCAR:
// 设置软件载波检测标志(1 - 开启; 0 - 关闭)。
200         return -EINVAL; /* not implemented */ /* 未实现 */
201     default:
202         return -EINVAL;
203 }
204 }
205

```

7.9.3 其它信息

7.9.3.1 波特率与波特率因子

波特率 = 1.8432MHz / (16 * 波特率因子)。程序中波特率与波特率因子的对应关系见表 7-10 所示。

表 7-10 波特率与波特率因子对应表

波特率	波特率因子		波特率	波特率因子	
	MSB,LSB	合并值		MSB,LSB	合并值
50	0x09,0x00	2304	1200	0x00,0x60	96
75	0x06,0x00	1536	1800	0x00,0x40	64
110	0x04,0x17	1047	2400	0x00,0x30	48
134.5	0x03,0x59	857	4800	0x00,0x18	24
150	0x03,0x00	768	9600	0x00,0x1c	12
200	0x02,0x40	576	19200	0x00,0x06	6
300	0x01,0x80	384	38400	0x00,0x03	3
600	0x00,0xc0	192			

