程序 10-2 linux/kernel/chr_drv/console.c

```
/*
 *  linux/kernel/console.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      console.c
 *
 * This module implements the console io functions
 *      'void con_init(void)'
 *      'void con_write(struct tty_queue * queue)'
 * Hopefully this will be a rather complete VT102 implementation.
 *
 * Beeping thanks to John T Kohl.
 *
 * Virtual Consoles, Screen Blanking, Screen Dumping, Color, Graphics
 *   Chars, and VT100 enhancements by Peter MacDonald.
 */
/*
 *      console.c
 *
 * 该模块实现控制台输入输出功能
 *      'void con_init(void)'
 *      'void con_write(struct tty_queue * queue)'
 * 希望这是一个非常完整的 VT102 实现。
 *
 * 感谢 John T Kohl 实现了蜂鸣指示子程序。
 *
 * 虚拟控制台、屏幕黑屏处理、屏幕拷贝、彩色处理、图形字符显示以及
 * VT100 终端增强操作由 Peter MacDonald 编制。
 */

/*
 *  NOTE!!! We sometimes disable and enable interrupts for a short while
 * (to put a word in video IO), but this will work even for keyboard
 * interrupts. We know interrupts aren't enabled when getting a keyboard
 * interrupt, as we use trap-gates. Hopefully all is well.
 */
/*
 * 注意!!! 我们有时短暂地禁止和允许中断（当输出一个字(word) 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，所以我们知道在处理
 * 一个键盘中断过程期间中断是被禁止的。希望一切均正常。
 */

/*
 * Code to check for different video-cards mostly by Galen Hunt,
 * <g-hunt@ee.utah.edu>
 */
/*
 * 检测不同显示卡的大多数代码是 Galen Hunt 编写的，
 * <g-hunt@ee.utah.edu>
```

```
 */

#include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据等。
#include <linux/tty.h>    // tty 头文件，定义有关 tty_io，串行通信方面的参数、常数。
#include <linux/config.h> // 内核配置头文件。定义硬盘类型（HD_TYPE）可选项。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。

#include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
#include <asm/system.h>   // 系统头文件。定义设置或修改描述符/中断门等的汇编宏。
#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。

#include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
#include <errno.h>        // 错误号头文件。包含系统中各种出错号。

// 该符号常量定义终端 IO 结构的默认数据。其中符号常数请参照 include/termios.h 文件。
#define DEF_TERMIOS \
(struct termios) { \
        ICRNL, \
        OPOST | ONLCR, \
        0, \
        IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, \
        0, \
        INIT_C_CC \
}


/*
 * These are set up by the setup-routine at boot-time:
 */
/*
 * 这些是 setup 程序在引导启动系统时设置的参数：
 */
// 参见对 boot/setup.s 的注释和 setup 程序读取并保留的系统参数表。

#define ORIG_X            (*(unsigned char *)0x90000)      // 初始光标列号。
#define ORIG_Y            (*(unsigned char *)0x90001)      // 初始光标行号。
#define ORIG_VIDEO_PAGE   (*(unsigned short *)0x90004)     // 初始显示页面。
#define ORIG_VIDEO_MODE   ((*(unsigned short *)0x90006) & 0xff)          // 显示模式。
#define ORIG_VIDEO_COLS   (((*(unsigned short *)0x90006) & 0xff00) >> 8) // 屏幕列数。
#define ORIG_VIDEO_LINES  ((*(unsigned short *)0x9000e) & 0xff)          // 屏幕行数。
#define ORIG_VIDEO_EGA_AX (*(unsigned short *)0x90008)     // [??]
#define ORIG_VIDEO_EGA_BX (*(unsigned short *)0x9000a)     // 显示内存大小和色彩模式。
#define ORIG_VIDEO_EGA_CX (*(unsigned short *)0x9000c)     // 显示卡特性参数。

// 定义显示器单色/彩色显示模式类型符号常数。
#define VIDEO_TYPE_MDA    0x10    /* Monochrome Text Display    */ /* 单色文本   */
#define VIDEO_TYPE_CGA    0x11    /* CGA Display                */ /* CGA 显示器  */
#define VIDEO_TYPE_EGAM   0x20    /* EGA/VGA in Monochrome Mode */ /* EGA/VGA 单色*/
#define VIDEO_TYPE_EGAC   0x21    /* EGA/VGA in Color Mode      */ /* EGA/VGA 彩色*/

#define NPAR 16                   // 转义字符序列中最大参数个数。

int NR_CONSOLES = 0;              // 系统实际支持的虚拟控制台数量。
```

```
78
79 extern void keyboard_interrupt(void);          // 键盘中断处理程序（keyboard.S）。
80
   // 以下这些静态变量是本文件函数中使用的一些全局变量。
   // video_type;              使用的显示类型；
   // video_num_columns;       屏幕文本列数；
   // video_mem_base;          物理显示内存基地址；
   // video_mem_term;          物理显示内存末端地址；
   // video_size_row;          屏幕每行使用的字节数；
   // video_num_lines;         屏幕文本行数；
   // video_page;              初试显示页面；
   // video_port_reg;          显示控制选择寄存器端口；
   // video_port_val;          显示控制数据寄存器端口。
81 static unsigned char    video_type;            /* Type of display being used   */
82 static unsigned long    video_num_columns;     /* Number of text columns       */
83 static unsigned long    video_mem_base;        /* Base of video memory         */
84 static unsigned long    video_mem_term;        /* End of video memory          */
85 static unsigned long    video_size_row;        /* Bytes per row                */
86 static unsigned long    video_num_lines;       /* Number of test lines         */
87 static unsigned char    video_page;            /* Initial video page           */
88 static unsigned short   video_port_reg;        /* Video register select port   */
89 static unsigned short   video_port_val;        /* Video register value port    */
90 static int can_do_colour = 0;                  // 标志：可使用彩色功能。
91
   // 虚拟控制台结构。其中包含一个虚拟控制台的当前所有信息。其中 vc_origin 和 vc_scr_end
   // 是当前正在处理的虚拟控制台执行快速滚屏操作时使用的起始行和末行对应的显示内存位置。
   // vc_video_mem_start 和 vc_video_mem_end 是当前虚拟控制台使用的显示内存区域部分。
   // vc -- Virtual Console。
92 static struct {
93         unsigned short  vc_video_erase_char;    // 擦除字符属性及字符（0x0720）
94         unsigned char   vc_attr;                // 字符属性。
95         unsigned char   vc_def_attr;            // 默认字符属性。
96         int             vc_bold_attr;           // 粗体字符属性。
97         unsigned long   vc_ques;                // 问号字符。
98         unsigned long   vc_state;               // 处理转义或控制序列的当前状态。
99         unsigned long   vc_restate;             // 处理转义或控制序列的下一状态。
100        unsigned long   vc_checkin;
101        unsigned long   vc_origin;              /* Used for EGA/VGA fast scroll */
102        unsigned long   vc_scr_end;             /* Used for EGA/VGA fast scroll */
103        unsigned long   vc_pos;                 // 当前光标对应的显示内存位置。
104        unsigned long   vc_x,vc_y;              // 当前光标列、行值。
105        unsigned long   vc_top,vc_bottom;       // 滚动时顶行行号；底行行号。
106        unsigned long   vc_npar,vc_par[NPAR];   // 转义序列参数个数和参数数组。
107        unsigned long   vc_video_mem_start;     /* Start of video RAM           */
108        unsigned long   vc_video_mem_end;       /* End of video RAM (sort of)   */
109        unsigned int    vc_saved_x;             // 保存的光标列号。
110        unsigned int    vc_saved_y;             // 保存的光标行号。
111        unsigned int    vc_iscolor;             // 彩色显示标志。
112        char *          vc_translate;           // 使用的字符集。
113 } vc_cons [MAX_CONSOLES];
114
   // 为了便于引用，以下定义当前正在处理控制台信息的符号。含义同上。其中 currcons 是使用
   // vc_cons[]结构的函数参数中的当前虚拟终端号。
```

```
115 #define origin          (vc_cons[currcons].vc_origin)  // 快速滚屏操作起始内存位置。
116 #define scr_end         (vc_cons[currcons].vc_scr_end) // 快速滚屏操作末端内存位置。
117 #define pos             (vc_cons[currcons].vc_pos)
118 #define top             (vc_cons[currcons].vc_top)
119 #define bottom          (vc_cons[currcons].vc_bottom)
120 #define x               (vc_cons[currcons].vc_x)
121 #define y               (vc_cons[currcons].vc_y)
122 #define state           (vc_cons[currcons].vc_state)
123 #define restate         (vc_cons[currcons].vc_restate)
124 #define checkin         (vc_cons[currcons].vc_checkin)
125 #define npar            (vc_cons[currcons].vc_npar)
126 #define par             (vc_cons[currcons].vc_par)
127 #define ques            (vc_cons[currcons].vc_ques)
128 #define attr            (vc_cons[currcons].vc_attr)
129 #define saved_x         (vc_cons[currcons].vc_saved_x)
130 #define saved_y         (vc_cons[currcons].vc_saved_y)
131 #define translate       (vc_cons[currcons].vc_translate)
132 #define video_mem_start (vc_cons[currcons].vc_video_mem_start)  // 使用显存的起始位置。
133 #define video_mem_end   (vc_cons[currcons].vc_video_mem_end)    // 使用显存的末端位置。
134 #define def_attr        (vc_cons[currcons].vc_def_attr)
135 #define video_erase_char  (vc_cons[currcons].vc_video_erase_char)
136 #define iscolor         (vc_cons[currcons].vc_iscolor)
137
138 int blankinterval = 0;              // 设定的屏幕黑屏间隔时间。
139 int blankcount = 0;                 // 黑屏时间计数。
140
141 static void sysbeep(void);          // 系统蜂鸣函数。
142
143 /*
144  * this is what the terminal answers to a ESC-Z or csi0c
145  * query (= vt100 response).
146  */
    /*
     * 下面是终端回应 ESC-Z 或 csi0c 请求的应答（=vt100 响应）。
     */
    // csi - 控制序列引导码(Control Sequence Introducer)。
    // 主机通过发送不带参数或参数是 0 的设备属性（DA）控制序列（ 'ESC [c' 或 'ESC [0c' ）
    // 要求终端应答一个设备属性控制序列（ESC Z 的作用与此相同），终端则发送以下序列来响应
    // 主机。该序列（即 'ESC [?1;2c' ）表示终端是具有高级视频功能的 VT100 兼容终端。
147 #define RESPONSE "\033[?1;2c"
148
    // 定义使用的字符集。其中上半部分时普通 7 比特 ASCII 代码，即 US 字符集。下半部分对应
    // VT100 终端设备中的线条字符，即显示图表线条的字符集。
149 static char * translations[] = {
150 /* normal 7-bit ascii */
151         " !\"#$%&'()*+,-./0123456789:;<=>?"
152         "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_"
153         "`abcdefghijklmnopqrstuvwxyz{|}~ ",
154 /* vt100 graphics */
155         " !\"#$%&'()*+,-./0123456789:;<=>?"
156         "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^ "
157         "\004\261\007\007\007\007\370\361\007\007\275\267\326\323\327\304"
158         "\304\304\304\304\307\266\320\322\272\363\362\343\|\007\234\007 "
```

```
159 };
160
161 #define NORM_TRANS (translations[0])
162 #define GRAF_TRANS (translations[1])
163
    //// 跟踪光标当前位置。
    // 参数：currcons - 当前虚拟终端号；new_x - 光标所在列号；new_y - 光标所在行号。
    // 更新当前光标位置变量 x,y，并修正光标在显示内存中的对应位置 pos。该函数会首先检查
    // 参数的有效性。如果给定的光标列号超出显示器最大列数，或者光标行号不低于显示的最大
    // 行数，则退出。否则就更新当前光标变量和新光标位置对应在显示内存中位置 pos。
    // 注意，函数中的所有变量实际上是 vc_cons[currcons]结构中的相应字段。以下函数相同。
164 /* NOTE! gotoxy thinks x==video_num_columns is ok */
    /* 注意！gotoxy 函数认为 x==video_num_columns 时是正确的 */
165 static inline void gotoxy(int currcons, int new_x,unsigned int new_y)
166 {
167         if (new_x > video_num_columns || new_y >= video_num_lines)
168                 return;
169         x = new_x;
170         y = new_y;
171         pos = origin + y*video_size_row + (x<<1);    // 1 列用 2 个字节表示，所以 x<<1。
172 }
173
    //// 设置滚屏起始显示内存地址。
    // 再次提醒，函数中变量基本上都是 vc_cons[currcons] 结构中的相应字段。
174 static inline void set_origin(int currcons)
175 {
    // 首先判断显示卡类型。 对于 EGA/VGA 卡，我们可以指定屏内行范围（区域）进行滚屏操作，
    // 而 MDA 单色显示卡只能进行整屏滚屏操作。因此只有 EGA/VGA 卡才需要设置滚屏起始行显示
    // 内存地址（起始行是 origin 对应的行）。即显示类型如果不是 EGA/VGA 彩色模式，也不是
    // EGA/VGA 单色模式，那么就直接返回。另外，我们只对前台控制台进行操作，因此当前控制台
    // currcons 必须是前台控制台时，我们才需要设置其滚屏起始行对应的内存起点位置。
176         if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
177                 return;
178         if (currcons != fg_console)
179                 return;
    // 然后向显示寄存器选择端口 video_port_reg 输出 12，即选择显示控制数据寄存器 r12，接着
    // 写入滚屏起始地址高字节。其中向右移动 9 位，实际上表示向右移动 8 位再除以 2（屏幕上 1
    // 个字符用 2 字节表示）。再选择显示控制数据寄存器 r13，然后写入滚屏起始地址低字节。向
    // 右移动 1 位表示除以 2，同样代表屏幕上 1 个字符用 2 字节表示。输出值相对于默认显示内存
    // 起始位置 video_mem_base 进行操作，例如对于 EGA/VGA 彩色模式，viedo_mem_base = 物理
    // 内存地址 0xb8000。
180         cli();
181         outb_p(12, video_port_reg);     // 选择数据寄存器 r12，输出滚屏起始位置高字节。
182         outb_p(0xff&((origin-video_mem_base)>>9), video_port_val);
183         outb_p(13, video_port_reg);     // 选择数据寄存器 r13，输出滚屏起始位置低字节。
184         outb_p(0xff&((origin-video_mem_base)>>1), video_port_val);
185         sti();
186 }
187
    //// 向上卷动一行。
    // 将屏幕滚动窗口向下移动一行，并在屏幕滚动区域底出现的新行上添加空格字符。滚屏区域
    // 必须大于 1 行。参见程序列表后说明。
188 static void scrup(int currcons)
```

```
189 {
    // 滚屏区域必须起码有 2 行。如果滚屏区域顶行号大于等于区域底行号，则不满足进行滚行操作
    // 的条件。另外，对于 EGA/VGA 卡，我们可以指定屏内行范围（区域）进行滚屏操作，而 MDA 单
    // 色显示卡只能进行整屏滚屏操作。该函数对 EGA 和 MDA 显示类型进行分别处理。如果显示类型
    // 是 EGA，则还分为整屏窗口移动和区域内窗口移动。这里首先处理显示卡是 EGA/VGA 显示类型
    // 的情况。
190         if (bottom<=top)
191                 return;
192         if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
193         {
    // 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。于是把整个屏幕窗口左上角对应的起始内存位置 origin 调整为向下移一行对应的内存
    // 位置，同时也跟踪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。
    // 最后把新屏幕窗口内存起始位置值 origin 写入显示控制器中。
194                 if (!top && bottom == video_num_lines) {
195                         origin += video_size_row;
196                         pos += video_size_row;
197                         scr_end += video_size_row;
    // 如果屏幕窗口末端所对应的显示内存指针 scr_end 超出了实际显示内存末端，则将屏幕内容
    // 除第一行以外所有行对应的内存数据移动到显示内存的起始位置 video_mem_start 处，并在
    // 整屏窗口向下移动出现的新行上填入空格字符。然后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。
    // 这段嵌入汇编程序首先将（屏幕字符行数 - 1）行对应的内存数据移动到显示内存起始位置
    // video_mem_start 处，然后在随后的内存位置处添加一行空格（擦除）字符数据。
    // %0 -eax(擦除字符+属性)；%1 -ecx（(屏幕字符行数-1)所对应的字符数/2，以长字移动）；
    // %2 -edi(显示内存起始位置 video_mem_start)；%3 -esi(屏幕窗口内存起始位置 origin)。
    // 移动方向：[edi]➔[esi]，移动 ecx 个长字。
198                         if (scr_end > video_mem_end) {
199                                 __asm__("cld\n\t"               // 清方向位。
200                                         "rep\n\t"               // 重复操作，将当前屏幕内存
201                                         "movsl\n\t"             // 数据移动到显示内存起始处。
202                                         "movl _video_num_columns,%1\n\t"
203                                         "rep\n\t"               // 在新行上填入空格字符。
204                                         "stosw"
205                                         ::"a" (video_erase_char),
206                                         "c" ((video_num_lines-1)*video_num_columns>>1),
207                                         "D" (video_mem_start),
208                                         "S" (origin)
209                                         :"cx","di","si");
210                                 scr_end -= origin-video_mem_start;
211                                 pos -= origin-video_mem_start;
212                                 origin = video_mem_start;
    // 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(屏幕行数)；%2 - edi（最后 1 行开始处对应内存位置）；
213                         } else {
214                                 __asm__("cld\n\t"
215                                         "rep\n\t"               // 重复操作，在新出现行上
216                                         "stosw"                 // 填入擦除字符(空格字符)。
217                                         ::"a" (video_erase_char),
218                                         "c" (video_num_columns),
219                                         "D" (scr_end-video_size_row)
220                                         :"cx","di");
```

```
221                         }
    // 然后把新屏幕滚动窗口内存起始位置值 origin 写入显示控制器中。
222                         set_origin(currcons);
    // 否则表示不是整屏移动。即表示从指定行 top 开始到 bottom 区域中的所有行向上移动 1 行，
    // 指定行 top 被删除。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，并在最下面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 bottom 行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)。
223                 } else {
224                         __asm__("cld\n\t"
225                                 "rep\n\t"           // 循环操作，将 top+1 到 bottom 行
226                                 "movsl\n\t"         // 所对应的内存块移到 top 行开始处。
227                                 "movl _video_num_columns,%%ecx\n\t"
228                                 "rep\n\t"           // 在新行上填入擦除字符。
229                                 "stosw"
230                                 ::"a" (video_erase_char),
231                                 "c" ((bottom-top-1)*video_num_columns>>1),
232                                 "D" (origin+video_size_row*top),
233                                 "S" (origin+video_size_row*(top+1))
234                                 :"cx","di","si");
235                 }
236         }
    // 如果显示类型不是 EGA（而是 MDA ），则执行下面移动操作。因为 MDA 显示控制卡只能整屏滚
    // 动，并且会自动调整超出显示范围的情况，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。
237         else            /* Not EGA/VGA */
238         {
239                 __asm__("cld\n\t"
240                         "rep\n\t"
241                         "movsl\n\t"
242                         "movl _video_num_columns,%%ecx\n\t"
243                         "rep\n\t"
244                         "stosw"
245                         ::"a" (video_erase_char),
246                         "c" ((bottom-top-1)*video_num_columns>>1),
247                         "D" (origin+video_size_row*top),
248                         "S" (origin+video_size_row*(top+1))
249                         :"cx","di","si");
250         }
251 }
252
    //// 向下卷动一行。
    // 将屏幕滚动窗口向上移动一行，相应屏幕滚动区域内容向下移动 1 行。并在移动开始行的上
    // 方出现一新行。参见程序列表后说明。处理方法与 scrup()相似，只是为了在移动显示内存
    // 数据时不会出现数据覆盖的问题，复制操作是以逆向进行的，即先从屏幕倒数第 2 行的最后
    // 一个字符开始复制到最后一行，再将倒数第 3 行复制到倒数第 2 行等等。因为此时对 EGA/
    // VGA 显示类型和 MDA 类型的处理过程完全一样，所以该函数实际上没有必要写两段相同的代
    // 码。即这里 if 和 else 语句块中的操作完全一样！
253 static void scrdown(int currcons)
254 {
    // 同样，滚屏区域必须起码有 2 行。如果滚屏区域顶行号大于等于区域底行号，则不满足进行滚
    // 行操作的条件。另外，对于 EGA/VGA 卡，我们可以指定屏内行范围（区域）进行滚屏操作，而
    // MDA 单色显示卡只能进行整屏滚屏操作。由于窗口向上移动最多移动到当前控制台占用显示区
```

```
    // 域内存的起始位置，因此不会发生屏幕窗口末端所对应的显示内存指针 scr_end 超出实际显示
    // 内存末端的情况，所以这里只需要处理普通的内存数据移动情况。
255         if (bottom <= top)
256                 return;
257         if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
258         {
    // %0 - eax(擦除字符+属性)；%1 - ecx(top 行到 bottom-1 行的行数所对应的内存长字数)；
    // %2 - edi(窗口右下角最后一个长字位置)；%3 - esi(窗口倒数第 2 行最后一个长字位置)。
    // 移动方向：[esi]➔[edi]，移动 ecx 个长字。
259                 __asm__("std\n\t"               // 置方向位！！
260                         "rep\n\t"               // 重复操作，向下移动从 top 行到
261                         "movsl\n\t"             // bottom-1 行对应的内存数据。
262                         "addl $2,%%edi\n\t"     /* %edi has been decremented by 4 */
                                                    /* %edi 已减 4，因也是反向填擦除字符*/
263                         "movl _video_num_columns,%%ecx\n\t"
264                         "rep\n\t"               // 将擦除字符填入上方新行中。
265                         "stosw"
266                         ::"a" (video_erase_char),
267                         "c" ((bottom-top-1)*video_num_columns>>1),
268                         "D" (origin+video_size_row*bottom-4),
269                         "S" (origin+video_size_row*(bottom-1)-4)
270                         :"ax","cx","di","si");
271         }
    // 如果不是 EGA 显示类型，则执行以下操作（与上面完全一样）。
272         else            /* Not EGA/VGA */
273         {
274                 __asm__("std\n\t"
275                         "rep\n\t"
276                         "movsl\n\t"
277                         "addl $2,%%edi\n\t"     /* %edi has been decremented by 4 */
278                         "movl _video_num_columns,%%ecx\n\t"
279                         "rep\n\t"
280                         "stosw"
281                         ::"a" (video_erase_char),
282                         "c" ((bottom-top-1)*video_num_columns>>1),
283                         "D" (origin+video_size_row*bottom-4),
284                         "S" (origin+video_size_row*(bottom-1)-4)
285                         :"ax","cx","di","si");
286         }
287 }
288
    //// 光标在同列位置下移一行。
    // 如果光标没有处在最后一行上，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（加上一行字符所对应的内存长度）。否则就需要将屏幕窗口内容上移一行。
    // 函数名称 lf（line feed 换行）是指处理控制字符 LF。
289 static void lf(int currcons)
290 {
291         if (y+1<bottom) {
292                 y++;
293                 pos += video_size_row;          // 加上屏幕一行占用内存的字节数。
294                 return;
295         }
296         scrup(currcons);                        // 将屏幕窗口内容上移一行。
```

```
297 }
298
    //// 光标在同列上移一行。
    // 如果光标不在屏幕第一行上，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。否则需要将屏幕窗口内容下移一行。
    // 函数名称 ri（reverse index 反向索引）是指控制字符 RI 或转义序列“ESC M”。
299 static void ri(int currcons)
300 {
301         if (y>top) {
302                 y--;
303                 pos -= video_size_row;          // 减去屏幕一行占用内存的字节数。
304                 return;
305         }
306         scrdown(currcons);                      // 将屏幕窗口内容下移一行。
307 }
308
    // 光标回到第 1 列（0 列）。
    // 调整光标对应内存位置 pos。光标所在列号*2 即是 0 列到光标所在列对应的内存字节长度。
    // 函数名称 cr（carriage return 回车）指明处理的控制字符是回车字符。
309 static void cr(int currcons)
310 {
311         pos -= x<<1;                            // 减去 0 列到光标处占用的内存字节数。
312         x=0;
313 }
314
    // 擦除光标前一字符（用空格替代）（del - delete 删除）。
    // 如果光标没有处在 0 列，则将光标对应内存位置 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，并将光标所在位置处字符擦除。
315 static void del(int currcons)
316 {
317         if (x) {
318                 pos -= 2;
319                 x--;
320                 *(unsigned short *)pos = video_erase_char;
321         }
322 }
323
    //// 删除屏幕上与光标位置相关的部分。
    // ANSI 控制序列：’ESC [ Ps J’（Ps =0 -删除光标处到屏幕底端；1 -删除屏幕开始到光标处；
    // 2 - 整屏删除）。本函数根据指定的控制序列具体参数值，执行与光标位置相关的删除操作，
    // 并且在擦除字符或行时光标位置不变。
    // 函数名称 csi_J （CSI - Control Sequence Introducer，即控制序列引导码）指明对控制
    // 序列“CSI Ps J”进行处理。
    // 参数：vpar - 对应上面控制序列中 Ps 的值。
324 static void csi_J(int currcons, int vpar)
325 {
326         long count __asm__("cx");               // 设为寄存器变量。
327         long start __asm__("di");
328
    // 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。
329         switch (vpar) {
330                 case 0: /* erase from cursor to end of display */
331                         count = (scr_end-pos)>>1;    /* 擦除光标到屏幕底端所有字符 */
```

```
332                     start = pos;
333                     break;
334             case 1: /* erase from start to cursor */
335                     count = (pos-origin)>>1;  /* 删除从屏幕开始到光标处的字符 */
336                     start = origin;
337                     break;
338             case 2: /* erase whole display */     /* 删除整个屏幕上的所有字符 */
339                     count = video_num_columns * video_num_lines;
340                     start = origin;
341                     break;
342             default:
343                     return;
344     }
    // 然后使用擦除字符填写被删除字符的地方。
    // %0 -ecx(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）。
345     __asm__("cld\n\t"
346             "rep\n\t"
347             "stosw\n\t"
348             ::"c" (count),
349             "D" (start),"a" (video_erase_char)
350             :"cx","di");
351 }
352
    //// 删除一行上与光标位置相关的部分。
    // ANSI 转义字符序列：'ESC [ Ps K'（Ps = 0 删除到行尾；1 从开始删除；2 整行都删除）。
    // 本函数根据参数擦除光标所在行的部分或所有字符。擦除操作从屏幕上移走字符但不影响其
    // 他字符。擦除的字符被丢弃。在擦除字符或行时光标位置不变。
    // 参数：par - 对应上面控制序列中 Ps 的值。
353 static void csi_K(int currcons, int vpar)
354 {
355     long count __asm__("cx");          // 设置寄存器变量。
356     long start __asm__("di");
357
    // 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。
358     switch (vpar) {
359             case 0: /* erase from cursor to end of line */
360                     if (x>=video_num_columns)     /* 删除光标到行尾所有字符 */
361                             return;
362                     count = video_num_columns-x;
363                     start = pos;
364                     break;
365             case 1: /* erase from start of line to cursor */
366                     start = pos - (x<<1);          /* 删除从行开始到光标处 */
367                     count = (x<video_num_columns)?x:video_num_columns;
368                     break;
369             case 2: /* erase whole line */            /* 将整行字符全删除 */
370                     start = pos - (x<<1);
371                     count = video_num_columns;
372                     break;
373             default:
374                     return;
375     }
    // 然后使用擦除字符填写删除字符的地方。
```

```
    // %0 - ecx(删除字符数 count)；%1 -edi(删除操作开始地址)；%2 -eax（填入的擦除字符）。
376         __asm__("cld\n\t"
377                 "rep\n\t"
378                 "stosw\n\t"
379                 ::"c" (count),
380                 "D" (start),"a" (video erase char)
381                 :"cx","di");
382 }
383
    //// 设置显示字符属性。
    // ANSI 转义序列：'ESC [ Ps;Ps m'。Ps = 0 - 默认属性；1 - 粗体并增亮；4 - 下划线；
    // 5 - 闪烁；7 - 反显；22 - 非粗体；24 - 无下划线；25 - 无闪烁；27 - 正显；
    // 30--38 - 设置前景色彩；39 - 默认前景色（White）；40--48 - 设置背景色彩；
    // 49 - 默认背景色（Black）。
    // 该控制序列根据参数设置字符显示属性。以后所有发送到终端的字符都将使用这里指定的属
    // 性，直到再次执行本控制序列重新设置字符显示的属性。
384 void csi m(int currcons )
385 {
386         int i;
387
    // 一个控制序列中可以带有多个不同参数。参数存储在数组 par[]中。下面就根据接收到的参数
    // 个数 npar，循环处理各个参数 Ps。
    // 如果 Ps = 0，则把当前虚拟控制台随后显示的字符属性设置为默认属性 def_attr。初始化时
    // def_attr 已被设置成 0x07（黑底白字）。
    // 如果 Ps = 1，则把当前虚拟控制台随后显示的字符属性设置为粗体或增亮显示。 如果是彩色
    // 显示，则把字符属性或上 0x08 让字符高亮度显示；如果是单色显示，则让字符带下划线显示。
    // 如果 Ps = 4，则对彩色和单色显示进行不同的处理。若此时不是彩色显示方式，则让字符带
    // 下划线显示。如果是彩色显示，那么若原来 vc_bold_attr 不等于-1 时就复位其背景色；否则
    // 的话就把前景色取反。若取反后前景色与背景色相同，就把前景色增 1 而取另一种颜色。
388         for (i=0;i<=npar;i++)
389                 switch (par[i]) {
390                         case 0: attr=def_attr;break;  /* default */
391                         case 1: attr=(iscolor?attr|0x08:attr|0x0f);break;  /* bold */
392                         /*case 4: attr=attr|0x01;break;*/  /* underline */
393                         case 4: /* bold */
394                           if (!iscolor)
395                             attr |= 0x01;             // 单色则带下划线显示。
396                           else
397                           { /* check if forground == background */
398                             if (vc_cons[currcons].vc_bold_attr != -1)
399                               attr = (vc_cons[currcons].vc_bold_attr&0x0f)|(0xf0&(attr));
400                             else
401                             { short newattr = (attr&0xf0)|(0xf&(~attr));
402                               attr = ((newattr&0xf)==((attr>>4)&0xf)?
403                                  (attr&0xf0)|(((attr&0xf)+1)%0xf):
404                                  newattr);
405                             }
406                           }
407                           break;
    // 如果 Ps = 5，则把当前虚拟控制台随后显示的字符设置为闪烁，即把属性字节比特位 7 置 1。
    // 如果 Ps = 7，则把当前虚拟控制台随后显示的字符设置为反显，即把前景和背景色交换。
    // 如果 Ps = 22，则取消随后字符的高亮度显示（取消粗体显示）。
    // 如果 Ps = 24，则对于单色显示是取消随后字符的下划线显示，对于彩色显示则是取消绿色。
```

```
    // 如果 Ps = 25，则取消随后字符的闪烁显示。
    // 如果 Ps = 27，则取消随后字符的反显。
    // 如果 Ps = 39，则复位随后字符的前景色为默认前景色（白色）。
    // 如果 Ps = 49，则复位随后字符的背景色为默认背景色（黑色）。
408                         case 5: attr=attr|0x80;break;  /* blinking */
409                         case 7: attr=(attr<<4)|(attr>>4);break;  /* negative */
410                         case 22: attr=attr&0xf7;break; /* not bold */
411                         case 24: attr=attr&0xfe;break;  /* not underline */
412                         case 25: attr=attr&0x7f;break;  /* not blinking */
413                         case 27: attr=def_attr;break; /* positive image */
414                         case 39: attr=(attr & 0xf0)|(def_attr & 0x0f); break;
415                         case 49: attr=(attr & 0x0f)|(def_attr & 0xf0); break;
    // 当 Ps（par[i]）为其他值时，则是设置指定的前景色或背景色。如果 Ps = 30..37，则是设置
    // 前景色；如果 Ps=40..47，则是设置背景色。有关颜色值请参见程序后说明。
416                         default:
417                           if (!can_do_colour)
418                             break;
419                           iscolor = 1;
420                           if ((par[i]>=30) && (par[i]<=38))        // 设置前景色。
421                             attr = (attr & 0xf0) | (par[i]-30);
422                           else  /* Background color */
423                             if ((par[i]>=40) && (par[i]<=48))      // 设置背景色。
424                               attr = (attr & 0x0f) | ((par[i]-40)<<4);
425                             else
426                                 break;
427                 }
428 }
429
    //// 设置显示光标。
    // 根据光标对应显示内存位置 pos，设置显示控制器光标的显示位置。
430 static inline void set_cursor(int currcons)
431 {
    // 既然我们需要设置显示光标，说明有键盘操作，因此需要恢复进行黑屏操作的延时计数值。
    // 另外，显示光标的控制台必须是前台控制台，因此若当前处理的台号 currcons 不是前台控
    // 制台就立刻返回。
432         blankcount = blankinterval;           // 复位黑屏操作的计数值。
433         if (currcons != fg_console)
434                 return;
    // 然后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），接着
    // 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。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。
435         cli();
436         outb_p(14, video_port_reg);
437         outb_p(0xff&((pos-video_mem_base)>>9), video_port_val);
438         outb_p(15, video_port_reg);
439         outb_p(0xff&((pos-video_mem_base)>>1), video_port_val);
440         sti();
441 }
442
    // 隐藏光标。
    // 把光标设置到当前虚拟控制台窗口的末端，起到隐藏光标的作用。
443 static inline void hide_cursor(int currcons)
444 {
```

```
    // 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。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。
445         outb_p(14, video_port_reg);
446         outb_p(0xff&((scr_end-video_mem_base)>>9), video_port_val);
447         outb_p(15, video_port_reg);
448         outb_p(0xff&((scr_end-video_mem_base)>>1), video_port_val);
449 }
450
    //// 发送对 VT100 的响应序列。
    // 即为响应主机请求终端向主机发送设备属性（DA）。主机通过发送不带参数或参数是 0 的 DA
    // 控制序列（'ESC [ 0c' 或 'ESC Z'）要求终端发送一个设备属性（DA）控制序列，终端则发
    // 送 85 行上定义的应答序列（即 'ESC [?1;2c'）来响应主机的序列，该序列告诉主机本终端
    // 是具有高级视频功能的 VT100 兼容终端。处理过程是将应答序列放入读缓冲队列中，并使用
    // copy_to_cooked()函数处理后放入辅助队列中。
451 static void respond(int currcons, struct tty_struct * tty)
452 {
453         char * p = RESPONSE;                    // 定义在第 147 行上。
454
455         cli();
456         while (*p) {                            // 将应答序列放入读队列。
457                 PUTCH(*p,tty->read_q);          // 逐字符放入。include/linux/tty.h，46 行。
458                 p++;
459         }
460         sti();                                  // 转换成规范模式（放入辅助队列中）。
461         copy_to_cooked(tty);                    // tty_io.c，120 行。
462 }
463
    //// 在光标处插入一空格字符。
    // 把光标开始处的所有字符右移一格，并将擦除字符插入在光标所在处。
464 static void insert_char(int currcons)
465 {
466         int i=x;
467         unsigned short tmp, old = video_erase_char;     // 擦除字符（加属性）。
468         unsigned short * p = (unsigned short *) pos;    // 光标对应内存位置。
469
470         while (i++<video_num_columns) {
471                 tmp=*p;
472                 *p=old;
473                 old=tmp;
474                 p++;
475         }
476 }
477
    //// 在光标处插入一行。
    // 将屏幕窗口从光标所在行到窗口底的内容向下卷动一行。光标将处在新的空行上。
478 static void insert_line(int currcons)
479 {
480         int oldtop,oldbottom;
481
    // 首先保存屏幕窗口卷动开始行 top 和最后行 bottom 值，然后从光标所在行让屏幕内容向下
    // 滚动一行。最后恢复屏幕窗口卷动开始行 top 和最后行 bottom 的原来值。
482         oldtop=top;
```

```
483         oldbottom=bottom;
484         top=y;                              // 设置屏幕卷动开始行和结束行。
485         bottom = video_num_lines;
486         scrdown(currcons);                  // 从光标开始处，屏幕内容向下滚动一行。
487         top=oldtop;
488         bottom=oldbottom;
489 }
490
    //// 删除一个字符。
    // 删除光标处的一个字符，光标右边的所有字符左移一格。
491 static void delete_char(int currcons)
492 {
493         int i;
494         unsigned short * p = (unsigned short *) pos;
495
    // 如果光标的当前列位置 x 超出屏幕最右列，则返回。否则从光标右一个字符开始到行末所有
    // 字符左移一格。然后在最后一个字符处填入擦除字符。
496         if (x>=video_num_columns)
497                 return;
498         i = x;
499         while (++i < video_num_columns) {       // 光标右所有字符左移 1 格。
500                 *p = *(p+1);
501                 p++;
502         }
503         *p = video_erase_char;                  // 最后填入擦除字符。
504 }
505
    //// 删除光标所在行。
    // 删除光标所在的一行，并从光标所在行开始屏幕内容上卷一行。
506 static void delete_line(int currcons)
507 {
508         int oldtop,oldbottom;
509
    // 首先保存屏幕卷动开始行 top 和最后行 bottom 值，然后从光标所在行让屏幕内容向上滚动
    // 一行。最后恢复屏幕卷动开始行 top 和最后行 bottom 的原来值。
510         oldtop=top;
511         oldbottom=bottom;
512         top=y;                                  // 设置屏幕卷动开始行和最后行。
513         bottom = video_num_lines;
514         scrup(currcons);                        // 从光标开始处，屏幕内容向上滚动一行。
515         top=oldtop;
516         bottom=oldbottom;
517 }
518
    //// 在光标处插入 nr 个字符。
    // ANSI 转义字符序列：'ESC [ Pn @'。在当前光标处插入 1 个或多个空格字符。Pn 是插入的字
    // 符数。默认是 1。光标将仍然处于第 1 个插入的空格字符处。在光标与右边界的字符将右移。
    // 超过右边界的字符将被丢失。
    // 参数 nr = 转义字符序列中的参数 Pn。
519 static void csi_at(int currcons, unsigned int nr)
520 {
    // 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。然后循环插入指定个空格字符。
```

```
521         if (nr > video_num_columns)
522                 nr = video_num_columns;
523         else if (!nr)
524                 nr = 1;
525         while (nr--)
526                 insert_char(currcons);
527 }
528
    //// 在光标位置处插入 nr 行。
    // ANSI 转义字符序列：'ESC [ Pn L'。该控制序列在光标处插入 1 行或多行空行。操作完成后
    // 光标位置不变。当空行被插入时，光标以下滚动区域内的行向下移动。滚动出显示页的行就
    // 丢失。
    // 参数 nr = 转义字符序列中的参数 Pn。
529 static void csi_L(int currcons, unsigned int nr)
530 {
    // 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。
    // 然后循环插入指定行数 nr 的空行。
531         if (nr > video_num_lines)
532                 nr = video_num_lines;
533         else if (!nr)
534                 nr = 1;
535         while (nr--)
536                 insert_line(currcons);
537 }
538
    //// 删除光标处的 nr 个字符。
    // ANSI 转义序列：'ESC [ Pn P'。该控制序列从光标处删除 Pn 个字符。当一个字符被删除时，
    // 光标右所有字符都左移。这会在右边界处产生一个空字符。其属性应该与最后一个左移字符
    // 相同，但这里作了简化处理，仅使用字符的默认属性（黑底白字空格 0x0720）来设置空字符。
    // 参数 nr = 转义字符序列中的参数 Pn。
539 static void csi_P(int currcons, unsigned int nr)
540 {
    // 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。然后循环删除光标处指定字符数 nr。
541         if (nr > video_num_columns)
542                 nr = video_num_columns;
543         else if (!nr)
544                 nr = 1;
545         while (nr--)
546                 delete_char(currcons);
547 }
548
    //// 删除光标处的 nr 行。
    // ANSI 转义序列：'ESC [ Pn M'。该控制序列在滚动区域内，从光标所在行开始删除 1 行或多
    // 行。当行被删除时，滚动区域内的被删行以下的行会向上移动，并且会在最底行添加 1 空行。
    // 若 Pn 大于显示页上剩余行数，则本序列仅删除这些剩余行，并对滚动区域外不起作用。
    // 参数 nr = 转义字符序列中的参数 Pn。
549 static void csi_M(int currcons, unsigned int nr)
550 {
    // 如果删除的行数大于屏幕最多行数，则截为屏幕显示行数；若欲删除的行数 nr 为 0，则删除
    // 1 行。然后循环删除指定行数 nr。
551         if (nr > video_num_lines)
552                 nr = video_num_lines;
```

```
553         else if (!nr)
554                 nr=1;
555         while (nr--)
556                 delete_line(currcons);
557 }
558
    //// 保存当前光标位置。
559 static void save_cur(int currcons)
560 {
561         saved_x=x;
562         saved_y=y;
563 }
564
    //// 恢复保存的光标位置。
565 static void restore_cur(int currcons)
566 {
567         gotoxy(currcons,saved_x, saved_y);
568 }
569
570
    // 这个枚举定义用于下面 con_write()函数中处理转义序列或控制序列的解析。ESnormal 是初
    // 始进入状态，也是转义或控制序列处理完毕时的状态。
    // ESnormal -  表示处于初始正常状态。此时若接收到的是普通显示字符，则把字符直接显示
    //             在屏幕上；若接收到的是控制字符（例如回车字符），则对光标位置进行设置。
    //             当刚处理完一个转义或控制序列，程序也会返回到本状态。
    // ESesc    -  表示接收到转义序列引导字符 ESC（0x1b = 033 = 27）；如果在此状态下接收
    //             到一个'['字符，则说明转义序列引导码，于是跳转到 ESsquare 去处理。否则
    //             就把接收到的字符作为转义序列来处理。对于选择字符集转义序列'ESC (' 和
    //             'ESC )'，我们使用单独的状态 ESsetgraph 来处理；对于设备控制字符串序列
    //             'ESC P'，我们使用单独的状态 ESsetterm 来处理。
    // ESsquare -  表示已经接收到一个控制序列引导码（'ESC ['），表示接收到的是一个控制序
    //             列。于是本状态执行参数数组 par[]清零初始化工作。如果此时接收到的又是一
    //             个'['字符，则表示收到了'ESC [['序列。该序列是键盘功能键发出的序列，于
    //             是跳转到 Esfunckey 去处理。否则我们需要准备接收控制序列的参数，于是置
    //             状态 Esgetpars 并直接进入该状态去接收并保存序列的参数字符。
    // ESgetpars - 该状态表示我们此时要接收控制序列的参数值。参数用十进制数表示，我们把
    //             接收到的数字字符转换成数值并保存到 par[]数组中。如果收到一个分号 ';'，
    //             则还是维持在本状态，并把接收到的参数值保存在数据 par[]下一项中。若不是
    //             数字字符或分号，说明已取得所有参数，那么就转移到状态 ESgotpars 去处理。
    // ESgotpars - 表示我们已经接收到一个完整的控制序列。此时我们可以根据本状态接收到的结
    //             尾字符对相应控制序列进行处理。不过在处理之前，如果我们在 ESsquare 状态
    //             收到过 '?'，说明这个序列是终端设备私有序列。本内核不对支持对这种序列的
    //             处理，于是我们直接恢复到 ESnormal 状态。否则就去执行相应控制序列。待序
    //             列处理完后就把状态恢复到 ESnormal。
    // ESfunckey - 表示我们接收到了键盘上功能键发出的一个序列。我们不用显示。于是恢复到正
    //             常状态 ESnormal。
    // ESsetterm - 表示处于设备控制字符串序列状态（DCS）。此时若收到字符 'S'，则恢复初始
    //             的显示字符属性。若收到的字符是'L'或'l'，则开启或关闭折行显示方式。
    // ESsetgraph -表示收到设置字符集转移序列'ESC (' 或 'ESC )'。它们分别用于指定 G0 和 G1
    //             所用的字符集。此时若收到字符 '0'，则选择图形字符集作为 G0 和 G1，若收到
    //             的字符是 'B'，这选择普通 ASCII 字符集作为 G0 和 G1 的字符集。
571 enum { ESnormal, ESesc, ESsquare, ESgetpars, ESgotpars, ESfunckey,
572         ESsetterm, ESsetgraph };
```

```
573
    //// 控制台写函数。
    // 从终端对应的 tty 写缓冲队列中取字符，针对每个字符进行分析。若是控制字符或转义或控制
    // 序列，则进行光标定位、字符删除等的控制处理；对于普通字符就直接在光标处显示。
    // 参数 tty 是当前控制台使用的 tty 结构指针。
574 void con_write(struct tty_struct * tty)
575 {
576         int nr;
577         char c;
578         int currcons;
579
    // 该函数首先根据当前控制台使用的 tty 在 tty 表中的项位置取得对应的控制台号 currcons，
    // 然后计算出（CHARS()）目前 tty 写队列中含有的字符数 nr，并循环取出其中的每个字符进行
    // 处理。不过如果当前控制台由于接收到键盘或程序发出的暂停命令（如按键 Ctrl-S）而处于
    // 停止状态，那么本函数就停止处理写队列中的字符，退出函数。另外，如果取出的是控制字符
    // CAN（24）或 SUB（26），那么若是在转义或控制序列期间收到的，则序列不会执行而立刻终
    // 止，同时显示随后的字符。注意，con_write()函数只处理取队列字符数时写队列中当前含有
    // 的字符。这有可能在一个序列被放到写队列期间读取字符数，因此本函数前一次退出时 state
    // 有可能正处于处理转义或控制序列的其他状态上。
580         currcons = tty - tty_table;
581         if ((currcons>=MAX_CONSOLES) || (currcons<0))
582                 panic("con_write: illegal tty");
583
584         nr = CHARS(tty->write_q);               // 取写队列中字符数。在 tty.h 文件中。
585         while (nr--) {
586                 if (tty->stopped)
587                         break;
588                 GETCH(tty->write_q,c);          // 取 1 字符到 c 中。
589                 if (c == 24 || c == 26)         // 控制字符 CAN、SUB -  取消、替换。
590                         state = ESnormal;
591                 switch(state) {
    // 如果从写队列中取出的字符是普通显示字符代码，就直接从当前映射字符集中取出对应的显示
    // 字符，并放到当前光标所处的显示内存位置处，即直接显示该字符。然后把光标位置右移一个
    // 字符位置。具体地，如果字符不是控制字符也不是扩展字符，即(31<c<127)，那么，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。然
    // 后将字符 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。
592                         case ESnormal:
593                                 if (c>31 && c<127) {            // 是普通显示字符。
594                                         if (x>=video_num_columns) {     // 要换行？
595                                                 x -= video_num_columns;
596                                                 pos -= video_size_row;
597                                                 lf(currcons);
598                                         }
599                                         __asm__("movb %2,%%ah\n\t"      // 写字符。
600                                                 "movw %%ax,%1\n\t"
601                                                 ::"a" (translate[c-32]),
602                                                 "m" (*(short *)pos),
603                                                 "m" (attr)
604                                                 :"ax");
605                                         pos += 2;
606                                         x++;
    // 如果字符 c 是转义字符 ESC，则转换状态 state 到 ESesc（637 行）。
607                                 } else if (c==27)               // ESC - 转义控制字符。
```

```
608                                     state=ESesc;
    // 如果 c 是换行符 LF(10)，或垂直制表符 VT(11)，或换页符 FF(12)，则光标移动到下 1 行。
609                             else if (c==10 || c==11 || c==12)
610                                     lf(currcons);
    // 如果 c 是回车符 CR(13)，则将光标移动到头列（0 列）。
611                             else if (c==13)                 // CR - 回车。
612                                     cr(currcons);
    // 如果 c 是 DEL(127)，则将光标左边字符擦除(用空格字符替代)，并将光标移到被擦除位置。
613                             else if (c==ERASE_CHAR(tty))
614                                     del(currcons);
    // 如果 c 是 BS(backspace,8)，则将光标左移 1 格，并相应调整光标对应内存位置指针 pos。
615                             else if (c==8) {                // BS - 后退。
616                                     if (x) {
617                                             x--;
618                                             pos -= 2;
619                                     }
    // 如果字符 c 是水平制表符 HT(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，则将光标移到下一行上。
620                             } else if (c==9) {              // HT - 水平制表。
621                                     c=8-(x&7);
622                                     x += c;
623                                     pos += c<<1;
624                                     if (x>video_num_columns) {
625                                             x -= video_num_columns;
626                                             pos -= video_size_row;
627                                             lf(currcons);
628                                     }
629                                     c=9;
    // 如果字符 c 是响铃符 BEL(7)，则调用蜂鸣函数，是扬声器发声。
630                             } else if (c==7)                // BEL - 响铃。
631                                     sysbeep();
    // 如果 c 是控制字符 SO（14）或 SI（15），则相应选择字符集 G1 或 G0 作为显示字符集。
632                             else if (c == 14)               // SO - 换出，使用 G1。
633                                     translate = GRAF_TRANS;
634                             else if (c == 15)               // SI - 换进，使用 G0。
635                                     translate = NORM_TRANS;
636                             break;
    // 如果在 ESnormal 状态收到转义字符 ESC(0x1b = 033 = 27)，则转到本状态处理。该状态对 C1
    // 中控制字符或转义字符进行处理。处理完后默认的状态将是 ESnormal。
637                     case ESesc:
638                             state = ESnormal;
639                             switch (c)
640                             {
641                               case '[':             // ESC [ - 是 CSI 序列。
642                                     state=ESsquare;
643                                     break;
644                               case 'E':             // ESC E - 光标下移 1 行回 0 列。
645                                     gotoxy(currcons,0,y+1);
646                                     break;
647                               case 'M':             // ESC M - 光标下移 1 行。
648                                     ri(currcons);
649                                     break;
650                               case 'D':             // ESC D - 光标下移 1 行。
```

```
651                                     lf(currcons);
652                                     break;
653                             case 'Z':               // ESC Z - 设备属性查询。
654                                     respond(currcons,tty);
655                                     break;
656                             case '7':               // ESC 7 - 保存光标位置。
657                                     save_cur(currcons);
658                                     break;
659                             case '8':               // ESC 8 - 恢复保存的光标原位置。
660                                     restore_cur(currcons);
661                                     break;
662                             case '(':  case ')': // ESC (、ESC ) - 选择字符集。
663                                     state = ESsetgraph;
664                                     break;
665                             case 'P':               // ESC P - 设置终端参数。
666                                     state = ESsetterm;
667                                     break;
668                             case '#':               // ESC # - 修改整行属性。
669                                     state = -1;
670                                     break;
671                             case 'c':               // ESC c - 复位到终端初始设置。
672                                     tty->termios = DEF_TERMIOS;
673                                     state = restate = ESnormal;
674                                     checkin = 0;
675                                     top = 0;
676                                     bottom = video_num_lines;
677                                     break;
678                              /* case '>':   Numeric keypad */
679                              /* case '=':   Appl. keypad */
680                             }
681                             break;
    // 如果在状态 ESesc（是转义字符 ESC）时收到字符'['，则表明是 CSI 控制序列，于是转到状
    // 态 ESsequare 来处理。首先对 ESC 转义序列保存参数的数组 par[]清零，索引变量 npar 指向
    // 首项，并且设置我们开始处于取参数状态 ESgetpars。如果接收到的字符不是'?'，则直接转
    // 到状态 ESgetpars 去处理，若接收到的字符是'?'，说明这个序列是终端设备私有序列，后面
    // 会有一个功能字符。于是去读下一字符，再到状态 ESgetpars 去处理代码处。如果此时接收
    // 到的字符还是'['，那么表明收到了键盘功能键发出的序列，于是设置下一状态为 ESfunckey。
    // 否则直接进入 ESgetpars 状态继续处理。
682                     case ESsquare:
683                             for(npar=0;npar<NPAR;npar++)      // 初始化参数数组。
684                                     par[npar]=0;
685                             npar=0;
686                             state=ESgetpars;
687                             if (c =='[')  /* Function key */ // 'ESC [['是功能键。
688                             { state=ESfunckey;
689                               break;
690                             }
691                             if (ques=(c=='?'))
692                                     break;
    // 该状态表示我们此时要接收控制序列的参数值。参数用十进制数表示，我们把接收到的数字字
    // 符转换成数值并保存到 par[]数组中。如果收到一个分号 ';'，则还是维持在本状态，并把接
    // 收到的参数值保存在数据 par[]下一项中。若不是数字字符或分号，说明已取得所有参数，那
    // 么就转移到状态 ESgotpars 去处理。
```

```
693                         case ESgetpars:
694                                 if (c==';' && npar<NPAR-1) {
695                                         npar++;
696                                         break;
697                                 } else if (c>='0' && c<='9') {
698                                         par[npar]=10*par[npar]+c-'0';
699                                         break;
700                                 } else state=ESgotpars;
    // ESgotpars 状态表示我们已经接收到一个完整的控制序列。此时我们可以根据本状态接收到的
    // 结尾字符对相应控制序列进行处理。不过在处理之前，如果我们在 ESsquare 状态收到过'?'，
    // 说明这个序列是终端设备私有序列。本内核不支持对这种序列的处理，于是我们直接恢复到
    // ESnormal 状态。否则就去执行相应控制序列。待序列处理完后就把状态恢复到 ESnormal。
701                         case ESgotpars:
702                                 state = ESnormal;
703                                 if (ques)
704                                 { ques =0;
705                                   break;
706                                 }
707                                 switch(c) {
    // 如果 c 是字符'G'或'`'，则 par[]中第 1 个参数代表列号。若列号不为零，则将光标左移 1 格。
708                                         case 'G': case '`':  // CSI Pn G -光标水平移动。
709                                                 if (par[0]) par[0]--;
710                                                 gotoxy(currcons,par[0],y);
711                                                 break;
    // 如果 c 是'A'，则第 1 个参数代表光标上移的行数。若参数为 0 则上移 1 行。
712                                         case 'A':           // CSI Pn A - 光标上移。
713                                                 if (!par[0]) par[0]++;
714                                                 gotoxy(currcons,x,y-par[0]);
715                                                 break;
    // 如果 c 是'B'或'e'，则第 1 个参数代表光标下移的行数。若参数为 0 则下移 1 行。
716                                         case 'B': case 'e':  // CSI Pn B - 光标下移。
717                                                 if (!par[0]) par[0]++;
718                                                 gotoxy(currcons,x,y+par[0]);
719                                                 break;
    // 如果 c 是'C'或'a'，则第 1 个参数代表光标右移的格数。若参数为 0 则右移 1 格。
720                                         case 'C': case 'a':  // CSI Pn C - 光标右移。
721                                                 if (!par[0]) par[0]++;
722                                                 gotoxy(currcons,x+par[0],y);
723                                                 break;
    // 如果 c 是'D'，则第 1 个参数代表光标左移的格数。若参数为 0 则左移 1 格。
724                                         case 'D':           // CSI Pn D - 光标左移。
725                                                 if (!par[0]) par[0]++;
726                                                 gotoxy(currcons,x-par[0],y);
727                                                 break;
    // 如果 c 是'E'，则第 1 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 1 行。
728                                         case 'E':       // CSI Pn E - 光标下移回 0 列。
729                                                 if (!par[0]) par[0]++;
730                                                 gotoxy(currcons,0,y+par[0]);
731                                                 break;
    // 如果 c 是'F'，则第 1 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 1 行。
732                                         case 'F':       // CSI Pn F - 光标上移回 0 列。
733                                                 if (!par[0]) par[0]++;
734                                                 gotoxy(currcons,0,y-par[0]);
```

```
735                                             break;
    // 如果 c 是'd'，则第 1 个参数代表光标所需在的行号（从 0 计数）。
736                                     case 'd':       // CSI Pn d - 在当前列置行位置。
737                                             if (par[0]) par[0]--;
738                                             gotoxy(currcons,x,par[0]);
739                                             break;
    // 如果 c 是'H'或'f'，则第 1 个参数代表光标移到的行号，第 2 个参数代表光标移到的列号。
740                                     case 'H': case 'f':  // CSI Pn H - 光标定位。
741                                             if (par[0]) par[0]--;
742                                             if (par[1]) par[1]--;
743                                             gotoxy(currcons,par[1],par[0]);
744                                             break;
    // 如果字符 c 是'J'，则第 1 个参数代表以光标所处位置清屏的方式：
    // 序列：'ESC [ Ps J'（Ps=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）。
745                                     case 'J':       // CSI Pn J - 屏幕擦除字符。
746                                             csi_J(currcons,par[0]);
747                                             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。
    // 转义序列：'ESC [ Ps K'（Ps = 0 删除到行尾；1 从开始删除；2 整行都删除）。
748                                     case 'K':       // CSI Pn K - 行内擦除字符。
749                                             csi_K(currcons,par[0]);
750                                             break;
    // 如果字符 c 是'L'，表示在光标位置处插入 n 行（控制序列 'ESC [ Pn L'）。
751                                     case 'L':       // CSI Pn L - 插入行。
752                                             csi_L(currcons,par[0]);
753                                             break;
    // 如果字符 c 是'M'，表示在光标位置处删除 n 行（控制序列 'ESC [ Pn M'）。
754                                     case 'M':       // CSI Pn M - 删除行。
755                                             csi_M(currcons,par[0]);
756                                             break;
    // 如果字符 c 是'P'，表示在光标位置处删除 n 个字符（控制序列 'ESC [ Pn P'）。
757                                     case 'P':       // CSI Pn P - 删除字符。
758                                             csi_P(currcons,par[0]);
759                                             break;
    // 如果字符 c 是'@'，表示在光标位置处插入 n 个字符（控制序列 'ESC [ Pn @' ）。
760                                     case '@':       // CSI Pn @ - 插入字符。
761                                             csi_at(currcons,par[0]);
762                                             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。
    // 转义序列：'ESC [ Pn m'。n=0 正常显示；1 加粗；4 加下划线；7 反显；27 正常显示等。
763                                     case 'm':       // CSI Ps m - 设置显示字符属性。
764                                             csi_m(currcons);
765                                             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。
766                                     case 'r':       // CSI Pn;Pn r - 设置滚屏上下界。
767                                             if (par[0]) par[0]--;
768                                             if (!par[1]) par[1] = video_num_lines;
769                                             if (par[0] < par[1] &&
770                                                 par[1] <= video_num_lines) {
771                                                     top=par[0];
772                                                     bottom=par[1];
773                                             }
774                                             break;
```

```
    // 如果字符 c 是's'，则表示保存当前光标所在位置。
775                                     case 's':       // CSI s - 保存光标位置。
776                                             save_cur(currcons);
777                                             break;
    // 如果字符 c 是'u'，则表示恢复光标到原保存的位置处。
778                                     case 'u':       // CSI u - 恢复保存的光标位置。
779                                             restore_cur(currcons);
780                                             break;
    // 如果字符 c 是'l'或'b'，则分别表示设置屏幕黑屏间隔时间和设置粗体字符显示。此时参数数
    // 组中 par[1]和 par[2]是特征值，它们分别必须为 par[1]= par[0]+13；par[2]= par[0]+17。
    // 在这个条件下，如果 c 是字符'l'，那么 par[0]中是开始黑屏时说延迟的分钟数；如果 c 是
    // 字符'b'，那么 par[0]中是设置的粗体字符属性值。
781                                     case 'l': /* blank interval */
782                                     case 'b': /* bold attribute */
783                                             if (!((npar >= 2) &&
784                                             ((par[1]-13) == par[0]) &&
785                                             ((par[2]-17) == par[0])))
786                                               break;
787                                             if ((c=='l')&&(par[0]>=0)&&(par[0]<=60))
788                                             {
789                                               blankinterval = HZ*60*par[0];
790                                               blankcount = blankinterval;
791                                             }
792                                             if (c=='b')
793                                               vc_cons[currcons].vc_bold_attr
794                                                 = par[0];
795                                     }
796                                     break;
    // 状态 ESfunckey 表示我们接收到了键盘上功能键发出的一个序列。我们不用显示。于是恢复到
    // 正常状态 ESnormal。
797                             case ESfunckey:                 // 键盘功能键码。
798                                     state = ESnormal;
799                                     break;
    // 状态 ESsetterm 表示处于设备控制字符串序列状态（DCS）。此时若收到字符 'S'，则恢复初
    // 始的显示字符属性。若收到的字符是'L'或'l'，则开启或关闭折行显示方式。
800                             case ESsetterm:  /* Setterm functions. */
801                                     state = ESnormal;
802                                     if (c == 'S') {
803                                             def_attr = attr;
804                                             video_erase_char = (video_erase_char&0x0ff) |
                                                                (def_attr<<8);
805                                     } else if (c == 'L')
806                                             ; /*linewrap on*/
807                                     else if (c == 'l')
808                                             ; /*linewrap off*/
809                                     break;
    // 状态 ESsetgraph 表示收到设置字符集转移序列'ESC (' 或 'ESC )'。它们分别用于指定 G0 和
    // G1 所用的字符集。此时若收到字符'0'，则选择图形字符集作为 G0 和 G1，若收到的字符是'B'，
    // 则选择普通 ASCII 字符集作为 G0 和 G1 的字符集。
810                             case ESsetgraph:        // 'CSI ( 0'或'CSI ( B' - 选择字符集。
811                                     state = ESnormal;
812                                     if (c == '0')
813                                             translate = GRAF_TRANS;
```

```
814                                 else if (c == 'B')
815                                         translate = NORM_TRANS;
816                                 break;
817                         default:
818                                 state = ESnormal;
819                 }
820         }
821         set_cursor(currcons);   // 最后根据上面设置的光标位置，设置显示控制器中光标位置。
822 }
823
824 /*
825  *  void con_init(void);
826  *
827  * This routine initalizes console interrupts, and does nothing
828  * else. If you want the screen to clear, call tty_write with
829  * the appropriate escape-sequece.
830  *
831  * Reads the information preserved by setup.s to determine the current display
832  * type and sets everything accordingly.
833  */
    /*
     * void con_init(void);
     *
     * 这个子程序初始化控制台中断，其他什么都不做。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。
     */
834 void con_init(void)
835 {
836         register unsigned char a;
837         char *display_desc = "????";
838         char *display_ptr;
839         int currcons = 0;                       // 当前虚拟控制台号。
840         long base, term;
841         long video_memory;
842
    // 首先根据 setup.s 程序取得的系统硬件参数（见本程序第 60--68 行）初始化几个本函数专用
    // 的静态全局变量。
843         video_num_columns = ORIG_VIDEO_COLS;    // 显示器显示字符列数。
844         video_size_row = video_num_columns * 2; // 每行字符需使用的字节数。
845         video_num_lines = ORIG_VIDEO_LINES;     // 显示器显示字符行数。
846         video_page = ORIG_VIDEO_PAGE;           // 当前显示页面。
847         video_erase_char = 0x0720;              // 擦除字符（0x20 是字符，0x07 属性）。
848         blankcount = blankinterval;             // 默认的黑屏间隔时间（嘀嗒数）。
849
    // 然后根据显示模式是单色还是彩色分别设置所使用的显示内存起始位置以及显示寄存器索引
    // 端口号和显示寄存器数据端口号。如果获得的 BIOS 显示方式等于 7，则表示是单色显示卡。
850         if (ORIG_VIDEO_MODE == 7)               /* Is this a monochrome display? */
851         {
852                 video_mem_base = 0xb0000;       // 设置单显映像内存起始地址。
853                 video_port_reg = 0x3b4;         // 设置单显索引寄存器端口。
854                 video_port_val = 0x3b5;         // 设置单显数据寄存器端口。
```

```
    // 接着我们根据 BIOS 中断 int 0x10 功能 0x12 获得的显示模式信息，判断显示卡是单色显示卡
    // 还是彩色显示卡。若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。虽然 EGA 卡上有较多显示内存，但在单色方式下最多只
    // 能利用地址范围在 0xb0000--0xb8000 之间的显示内存。然后置显示器描述字符串为'EGAm'。
    // 并会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。
    // 注意，这里使用了 bx 在调用中断 int 0x10 前后是否被改变的方法来判断卡的类型。若 BL 在
    // 中断调用后值被改变，表示显示卡支持 Ah=12h 功能调用，是 EGA 或后推出来的 VGA 等类型的
    // 显示卡。若中断调用返回值未变，表示显示卡不支持这个功能，则说明是一般单色显示卡。
855                 if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
856                 {
857                         video_type = VIDEO_TYPE_EGAM;  // 设置显示类型（EGA 单色）。
858                         video_mem_term = 0xb8000;      // 设置显示内存末端地址。
859                         display_desc = "EGAm";         // 设置显示描述字符串。
860                 }
    // 如果 BX 寄存器的值等于 0x10，则说明是单色显示卡 MDA，仅有 8KB 显示内存。
861                 else
862                 {
863                         video_type = VIDEO_TYPE_MDA;   // 设置显示类型(MDA 单色)。
864                         video_mem_term = 0xb2000;      // 设置显示内存末端地址。
865                         display_desc = "*MDA";         // 设置显示描述字符串。
866                 }
867         }
    // 如果显示方式不为 7，说明是彩色显示卡。此时文本方式下所用显示内存起始地址为 0xb8000；
    // 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。
868         else                                /* If not, it is color. */
869         {
870                 can_do_colour = 1;                      // 设置彩色显示标志。
871                 video_mem_base = 0xb8000;               // 显示内存起始地址。
872                 video_port_reg  = 0x3d4;                // 设置彩色显示索引寄存器端口。
873                 video_port_val  = 0x3d5;                // 设置彩色显示数据寄存器端口。
    // 再判断显示卡类别。如果 BX 不等于 0x10，则说明是 EGA 显示卡，此时共有 32KB 显示内存可用
    // （0xb8000-0xc0000）。否则说明是 CGA 显示卡，只能使用 8KB 显示内存（0xb8000-0xba000）。
874                 if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
875                 {
876                         video_type = VIDEO_TYPE_EGAC;  // 设置显示类型（EGA 彩色）。
877                         video_mem_term = 0xc0000;      // 设置显示内存末端地址。
878                         display_desc = "EGAc";         // 设置显示描述字符串。
879                 }
880                 else
881                 {
882                         video_type = VIDEO_TYPE_CGA;   // 设置显示类型（CGA）。
883                         video_mem_term = 0xba000;      // 设置显示内存末端地址。
884                         display_desc = "*CGA";         // 设置显示描述字符串。
885                 }
886         }
    // 现在我们来计算当前显示卡内存上可以开设的虚拟控制台数量。硬件允许开设的虚拟控制台数
    // 量等于总显示内存量 video_memory 除以每个虚拟控制台占用的字节数。每个虚拟控制台占用的
    // 显示内存数等于屏幕显示行数 video_num_lines 乘上每行字符占有的字节数 video_size_row。
    // 如果硬件允许开设的虚拟控制台数量大于系统限定的最大数量 MAX_CONSOLES，就把虚拟控制台
    // 数量设置为 MAX_CONSOLES。若这样计算出的虚拟控制台数量为 0，则设置为 1（不可能吧！）。
    // 最后总显示内存数除以判断出的虚拟控制台数即得到每个虚拟控制台占用显示内存字节数。
887         video_memory = video_mem_term - video_mem_base;
```

```
888         NR_CONSOLES = video_memory / (video_num_lines * video_size_row);
889         if (NR_CONSOLES > MAX_CONSOLES)            // MAX_CONSOLES = 8。
890                 NR_CONSOLES = MAX_CONSOLES;
891         if (!NR_CONSOLES)
892                 NR_CONSOLES = 1;
893         video_memory /= NR_CONSOLES;               // 每个虚拟控制台占用显示内存字节数。
894
895         /* Let the user known what kind of display driver we are using */
            /* 初始化用于滚屏的变量（主要用于 EGA/VGA） */
896
    // 然后我们在屏幕的右上角显示描述字符串。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 1 行右端差 4 个字符处（每个字符需 2 个
    // 字节，因此减 8），然后循环复制字符串的字符，并且每复制 1 个字符都空开 1 个属性字节。
897         display_ptr = ((char *)video_mem_base) + video_size_row - 8;
898         while (*display_desc)
899         {
900                 *display_ptr++ = *display_desc++;
901                 display_ptr++;
902         }
903
904         /* 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) */
905
    // 注意，此时当前虚拟控制台号 currcons 已被初始化位 0。因此下面实际上是初始化 0 号虚拟控
    // 制台的结构 vc_cons[0]中的所有字段值。例如，这里符号 origin 在前面第 115 行上已被定义为
    // vc_cons[0].vc_origin。下面首先设置 0 号控制台的默认滚屏开始内存位置 video_mem_start
    // 和默认滚屏末行内存位置，实际上它们也就是 0 号控制台占用的部分显示内存区域。然后初始
    // 设置 0 号虚拟控制台的其他属性和标志值。
906         base = origin = video_mem_start = video_mem_base;  // 默认滚屏开始内存位置。
907         term = video_mem_end = base + video_memory;        // 0 号屏幕内存末端位置。
908         scr_end = video_mem_start + video_num_lines * video_size_row; // 滚屏末端位置。
909         top     = 0;                               // 初始设置滚动时顶行行号和底行行号。
910         bottom  = video_num_lines;
911         attr = 0x07;                               // 初始设置显示字符属性（黑底白字）。
912         def_attr = 0x07;                           // 设置默认显示字符属性。
913         restate = state = ESnormal;                // 初始化转义序列操作的当前和下一状态。
914         checkin = 0;
915         ques = 0;                                  // 收到问号字符标志。
916         iscolor = 0;                               // 彩色显示标志。
917         translate = NORM_TRANS;                    // 使用的字符集（普通 ASCII 码表）。
918         vc_cons[0].vc_bold_attr = -1;              // 粗体字符属性标志（-1 表示不用）。
919
    // 在设置了 0 号控制台当前光标所在位置和光标对应的内存位置 pos 后，我们循环设置其余的几
    // 个虚拟控制台结构的参数值。除了各自占用的显示内存开始和结束位置不同，它们的初始值基
    // 本上都与 0 号控制台相同。
920         gotoxy(currcons,ORIG_X,ORIG_Y);
921         for (currcons = 1; currcons<NR_CONSOLES; currcons++) {
922                 vc_cons[currcons] = vc_cons[0];     // 复制 0 号结构的参数。
923                 origin = video_mem_start = (base += video_memory);
924                 scr_end = origin + video_num_lines * video_size_row;
925                 video_mem_end = (term += video_memory);
926                 gotoxy(currcons,0,0);               // 光标都初始化在屏幕左上角位置。
927         }
```

```
    // 最后设置当前前台控制台的屏幕原点（左上角）位置和显示控制器中光标显示位置，并设置键
    // 盘中断 0x21 陷阱门描述符（&keyboard_interrupt 是键盘中断处理过程地址）。然后取消中断
    // 控制芯片 8259A 中对键盘中断的屏蔽，允许响应键盘发出的 IRQ1 请求信号。最后复位键盘控
    // 制器以允许键盘开始正常工作。
928         update_screen();                        // 更新前台原点和设置光标位置。
929         set_trap_gate(0x21,&keyboard_interrupt); // 参见 system.h，第 36 行开始。
930         outb_p(inb_p(0x21)&0xfd,0x21);          // 取消对键盘中断的屏蔽，允许 IRQ1。
931         a=inb_p(0x61);                          // 读取键盘端口 0x61（8255A 端口 PB）。
932         outb_p(a|0x80,0x61);                    // 设置禁止键盘工作（位 7 置位），
933         outb_p(a,0x61);                         // 再允许键盘工作，用以复位键盘。
934 }
935
    // 更新当前前台控制台。
    // 把前台控制台转换为 fg_console 指定的虚拟控制台。fg_console 是设置的前台虚拟控制台号。
936 void update_screen(void)
937 {
938         set_origin(fg_console);                 // 设置滚屏起始显示内存地址。
939         set_cursor(fg_console);                 // 设置显示控制器中光标显示内存位置。
940 }
941
942 /* from bsd-net-2: */
943
    //// 停止蜂鸣。
    // 复位 8255A PB 端口的位 1 和位 0。参见 kernel/sched.c 程序后的定时器编程说明。
944 void sysbeepstop(void)
945 {
946         /* disable counter 2 */ /* 禁止定时器 2 */
947         outb(inb_p(0x61)&0xFC, 0x61);
948 }
949
950 int beepcount = 0;              // 蜂鸣时间嘀嗒计数。
951
    // 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（0x61）位 1 和 位 0（置位），然后设置定时器 2 通道发送一定的定时频率即可。
    // 参见 boot/setup.s 程序后 8259A 芯片编程方法和 kernel/sched.c 程序后的定时器编程说明。
952 static void sysbeep(void)
953 {
954         /* enable counter 2 */          /* 开启定时器 2 */
955         outb_p(inb_p(0x61)|3, 0x61);
956         /* set command for counter 2, 2 byte write */    /* 送设置定时器 2 命令 */
957         outb_p(0xB6, 0x43);             // 定时器芯片控制字寄存器端口。
958         /* send 0x637 for 750 HZ */     /* 设置频率为 750HZ，因此送定时值 0x637 */
959         outb_p(0x37, 0x42);             // 通道 2 数据端口分别送计数高低字节。
960         outb(0x06, 0x42);
961         /* 1/8 second */                /* 蜂鸣时间为 1/8 秒 */
962         beepcount = HZ/8;
963 }
964
    //// 拷贝屏幕。
    // 把屏幕内容复制到参数指定的用户缓冲区 arg 中。
    // 参数 arg 有两个用途，一是用于传递控制台号，二是作为用户缓冲区指针。
```

```
965 int do_screendump(int arg)
966 {
967         char *sptr, *buf = (char *)arg;
968         int currcons, l;
969
    // 函数首先验证用户提供的缓冲区容量，若不够则进行适当扩展。然后从其开始处取出控制台
    // 号 currcons。在判断控制台号有效之后，就把该控制台屏幕的所有内存内容复制到用户缓冲
    // 区中。
970         verify_area(buf,video_num_columns*video_num_lines);
971         currcons = get_fs_byte(buf);
972         if ((currcons<1) || (currcons>NR_CONSOLES))
973                 return -EIO;
974         currcons--;
975         sptr = (char *) origin;
976         for (l=video_num_lines*video_num_columns; l>0 ; l--)
977                 put_fs_byte(*sptr++,buf++);
978         return(0);
979 }
980
    // 黑屏处理。
    // 当用户在 blankInterval 时间间隔内没有按任何按键时就让屏幕黑屏，以保护屏幕。
981 void blank_screen()
982 {
983         if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
984                 return;
985 /* blank here. I can't find out how to do it, though */
986 }
987
    // 恢复黑屏的屏幕。
    // 当用户按下任何按键时，就恢复处于黑屏状态的屏幕显示内容。
988 void unblank_screen()
989 {
990         if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
991                 return;
992 /* unblank here */
993 }
994
    //// 控制台显示函数。
    // 该函数仅用于内核显示函数 printk()（kernel/printk.c），用于在当前前台控制台上显示
    // 内核信息。处理方法是循环取出缓冲区中的字符，并根据字符的特性控制光标移动或直接显
    // 示在屏幕上。
    // 参数 b 是 null 结尾的字符串缓冲区指针。
995 void console_print(const char * b)
996 {
997         int currcons = fg_console;
998         char c;
999
    // 循环读取缓冲区 b 中的字符。如果当前字符 c 是换行符，则对光标执行回车换行操作；然后
    // 去处理下一个字符。如果是回车符，就直接执行回车动作。然后去处理下一个字符。
1000         while (c = *(b++)) {
1001                 if (c == 10) {
1002                         cr(currcons);
1003                         lf(currcons);
```

```
1004                        continue;
1005                }
1006                if (c == 13) {
1007                        cr(currcons);
1008                        continue;
1009                }
    // 在读取了一个不是回车或换行字符后，如果发现当前光标列位置 x 已经到达屏幕右末端，则让
    // 光标折返到下一行开始处。然后把字符放到光标所处显示内存位置处，即在屏幕上显示出来。
    // 再把光标右移一格位置，为显示下一个字符作准备。
1010                if (x>=video_num_columns) {
1011                        x -= video_num_columns;
1012                        pos -= video_size_row;
1013                        lf(currcons);
1014                }
    // 寄存器 al 中是需要显示的字符，这里把属性字节放到 ah 中，然后把 ax 内容存储到光标内存
    // 位置 pos 处，即在光标处显示字符。
1015                __asm__("movb %2,%%ah\n\t"          // 属性字节放到 ah 中。
1016                        "movw %%ax,%1\n\t"          // ax 内容放到 pos 处。
1017                        ::"a" (c),
1018                        "m" (*(short *)pos),
1019                        "m" (attr)
1020                        :"ax");
1021                pos += 2;
1022                x++;
1023        }
1024        set_cursor(currcons); // 最后设置的光标内存位置，设置显示控制器中光标位置。
1025 }
1026
```