```
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。
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，定义任务结构 task_struct、任务 0 的数据等。
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，定义有关 tty_io、串行通信方面参数、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。定义设置或修改描述符/中断门等的嵌入式汇编宏。
17
   // 根据进程组号 pgrp 取得进程组所属的会话号。定义在 kernel/exit.c，161 行。
18 extern int session_of_pgrp(int pgrp);
   // 向使用指定 tty 终端的进程组中所有进程发送信号。定义在 chr_drv/tty_io.c，246 行。
19 extern int tty_signal(int sig, struct tty_struct *tty);
20
   // 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说明。
   // 例如波特率是 2400bps 时，对应的因子是 48（0x30）；9600bps 的因子是 12（0x1c）。
21 static unsigned short quotient[] = {
22         0, 2304, 1536, 1047, 857,
23         768, 576, 384, 192, 96,
24         64, 48, 24, 12, 6, 3
25 };
26
   //// 修改传输波特率。
   // 参数：tty - 终端对应的 tty 数据结构。
   // 在除数锁存标志 DLAB 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。写完后再复位 DLAB 位。对于串口 2，这两个端口分别是 0x2f8 和 0x2f9。
27 static void change_speed(struct tty_struct * tty)
28 {
29         unsigned short port,quot;
30
   // 函数首先检查参数 tty 指定的终端是否是串行终端，若不是则退出。对于串口终端的 tty 结
   // 构，其读缓冲队列 data 字段存放着串行端口基址（0x3f8 或 0x2f8），而一般控制台终端的
   // tty 结构的 read_q.data 字段值为 0。然后从终端 termios 结构的控制模式标志集中取得已设
   // 置的波特率索引号，并据此从波特率因子数组 quotient[]中取得对应的波特率因子值 quot。
   // CBAUD 是控制模式标志集中波特率位屏蔽码。
31         if (!(port = tty->read_q->data))
32                 return;
33         quot = quotient[tty->termios.c_cflag & CBAUD];
   // 接着把波特率因子 quot 写入串行端口对应 UART 芯片的波特率因子锁存器中。在写之前我们
   // 先要把线路控制寄存器 LCR 的除数锁存访问比特位 DLAB（位 7）置 1。然后把 16 位的波特
   // 率因子低高字节分别写入端口 0x3f8、0x3f9 （分别对应波特率因子低、高字节锁存器）。
   // 最后再复位 LCR 的 DLAB 标志位。
34         cli();
35         outb_p(0x80,port+3);        /* set DLAB */    // 首先设置除数锁定标志 DLAB。
```

```
36          outb_p(quot & 0xff,port);    /* LS of divisor */   // 输出因子低字节。
37          outb_p(quot >> 8,port+1);    /* MS of divisor */   // 输出因子高字节。
38          outb(0x03,port+3);           /* reset DLAB */      // 复位 DLAB。
39          sti();
40  }
41
    //// 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的目的。
42  static void flush(struct tty_queue * queue)
43  {
44          cli();
45          queue->head = queue->tail;
46          sti();
47  }
48
    //// 等待字符发送出去。
49  static void wait_until_sent(struct tty_struct * tty)
50  {
51          /* do nothing - not implemented */  /* 什么都没做 - 还未实现 */
52  }
53
    //// 发送 BREAK 控制符。
54  static void send_break(struct tty_struct * tty)
55  {
56          /* do nothing - not implemented */  /* 什么都没做 - 还未实现 */
57  }
58
    //// 取终端 termios 结构信息。
    // 参数：tty - 指定终端的 tty 结构指针；termios - 存放 termios 结构的用户缓冲区。
59  static int get_termios(struct tty_struct * tty, struct termios * termios)
60  {
61          int i;
62
    // 首先验证用户缓冲区指针所指内存区容量是否足够，如不够则分配内存。然后复制指定终端
    // 的 termios 结构信息到用户缓冲区中。最后返回 0。
63          verify_area(termios, sizeof (*termios));        // kernel/fork.c，24 行。
64          for (i=0 ; i< (sizeof (*termios)) ; i++)
65                  put_fs_byte( ((char *)&tty->termios)[i] , i+(char *)termios );
66          return 0;
67  }
68
    //// 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。
69  static int set_termios(struct tty_struct * tty, struct termios * termios,
70                         int channel)
71  {
72          int i, retsig;
73
74          /* If we try to set the state of terminal and we're not in the
75             foreground, send a SIGTTOU.  If the signal is blocked or
76             ignored, go ahead and perform the operation.  POSIX 7.2) */
          /* 如果试图设置终端的状态但此时终端不在前台，那么我们就需要发送
```

```
                一个 SIGTTOU 信号。如果该信号被进程屏蔽或者忽略了，就直接执行
                本次操作。  POSIX 7.2  */
    // 如果当前进程使用的 tty 终端的进程组号与进程的进程组号不同，即当前进程终端不在前台，
    // 表示当前进程试图修改不受控制的终端的 termios 结构。因此根据 POSIX 标准的要求这里需
    // 要发送 SIGTTOU 信号让使用这个终端的进程先暂时停止执行，以让我们先修改 termios 结构。
    // 如果发送信号函数 tty_signal()返回值是 ERESTARTSYS 或 EINTR，则等一会再执行本次操作。
77          if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
78                  retsig = tty_signal(SIGTTOU, tty);      // chr_drv/tty_io.c，246 行。
79                  if (retsig == -ERESTARTSYS || retsig == -EINTR)
80                          return retsig;
81          }
    // 接着把用户数据区中 termios 结构信息复制到指定终端 tty 结构的 termios 结构中。因为用
    // 户有可能已修改了终端串行口传输波特率，所以这里再根据 termios 结构中的控制模式标志
    // c_cflag 中的波特率信息修改串行 UART 芯片内的传输波特率。最后返回 0。
82          for (i=0 ; i< (sizeof (*termios)) ; i++)
83                  ((char *)&tty->termios)[i]=get_fs_byte(i+(char *)termios);
84          change_speed(tty);
85          return 0;
86  }
87
    //// 读取 termio 结构中的信息。
    // 参数：tty - 指定终端的 tty 结构指针；termio - 保存 termio 结构信息的用户缓冲区。
88  static int get_termio(struct tty_struct * tty, struct termio * termio)
89  {
90          int i;
91          struct termio tmp_termio;
92
    // 首先验证用户的缓冲区指针所指内存区容量是否足够，如不够则分配内存。然后将 termios
    // 结构的信息复制到临时 termio 结构中。这两个结构基本相同，但输入、输出、控制和本地
    // 标志集数据类型不同。前者的是 long，而后者的是 short。因此先复制到临时 termio 结构
    // 中目的是为了进行数据类型转换。
93          verify_area(termio, sizeof (*termio));
94          tmp_termio.c_iflag = tty->termios.c_iflag;
95          tmp_termio.c_oflag = tty->termios.c_oflag;
96          tmp_termio.c_cflag = tty->termios.c_cflag;
97          tmp_termio.c_lflag = tty->termios.c_lflag;
98          tmp_termio.c_line = tty->termios.c_line;
99          for(i=0 ; i < NCC ; i++)
100                 tmp_termio.c_cc[i] = tty->termios.c_cc[i];
    // 最后逐字节地把临时 termio 结构中的信息复制到用户 termio 结构缓冲区中。并返回 0。
101         for (i=0 ; i< (sizeof (*termio)) ; i++)
102                 put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio );
103         return 0;
104 }
105
106 /*
107  * This only works as the 386 is low-byt-first
108  */
    /*
     * 下面 termio 设置函数仅适用于低字节在前的 386 CPU。
     */
    //// 设置终端 termio 结构信息。
    // 参数：tty - 指定终端的 tty 结构指针；termio - 用户数据区中 termio 结构。
```

```
    // 将用户缓冲区 termio 的信息复制到终端的 termios 结构中。返回 0 。
109 static int set_termio(struct tty_struct * tty, struct termio * termio,
110                        int channel)
111 {
112         int i, retsig;
113         struct termio tmp_termio;
114
    // 与 set_termios()一样，如果进程使用的终端的进程组号与进程的进程组号不同，即当前进
    // 程终端不在前台，表示当前进程试图修改不受控制的终端的 termios 结构。因此根据 POSIX
    // 标准的要求这里需要发送 SIGTTOU 信号让使用这个终端的进程先暂时停止执行，以让我们先
    // 修改 termios 结构。如果发送信号函数 tty_signal()返回值是 ERESTARTSYS 或 EINTR，则等
    // 一会再执行本次操作。
115         if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
116                 retsig = tty_signal(SIGTTOU, tty);
117                 if (retsig == -ERESTARTSYS || retsig == -EINTR)
118                         return retsig;
119         }
    // 接着复制用户数据区中 termio 结构信息到临时 termio 结构中。然后再将 termio 结构的信息
    // 复制到 tty 的 termios 结构中。这样做的目的是为了对其中模式标志集的类型进行转换，即
    // 从 termio 的短整数类型转换成 termios 的长整数类型。但两种结构的 c_line 和 c_cc[]字段
    // 是完全相同的。
120         for (i=0 ; i< (sizeof (*termio)) ; i++)
121                 ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
122         *(unsigned short *)&tty->termios.c_iflag = tmp_termio.c_iflag;
123         *(unsigned short *)&tty->termios.c_oflag = tmp_termio.c_oflag;
124         *(unsigned short *)&tty->termios.c_cflag = tmp_termio.c_cflag;
125         *(unsigned short *)&tty->termios.c_lflag = tmp_termio.c_lflag;
126         tty->termios.c_line = tmp_termio.c_line;
127         for(i=0 ; i < NCC ; i++)
128                 tty->termios.c_cc[i] = tmp_termio.c_cc[i];
    // 最后因为用户有可能已修改了终端串行口传输波特率，所以这里再根据 termios 结构中的控制
    // 模式标志 c_cflag 中的波特率信息修改串行 UART 芯片内的传输波特率，并返回 0。
129         change_speed(tty);
130         return 0;
131 }
132
    //// tty 终端设备输入输出控制函数。
    // 参数：dev - 设备号；cmd - ioctl 命令；arg - 操作参数指针。
    // 该函数首先根据参数给出的设备号找出对应终端的 tty 结构，然后根据控制命令 cmd 分别进行
    // 处理。
133 int tty_ioctl(int dev, int cmd, int arg)
134 {
135         struct tty_struct * tty;
136         int     pgrp;
137
    // 首先根据设备号取得 tty 子设备号，从而取得终端的 tty 结构。若主设备号是 5（控制终端），
    // 则进程的 tty 字段即是 tty 子设备号。此时如果进程的 tty 子设备号是负数，表明该进程没有
    // 控制终端，即不能发出该 ioctl 调用，于是显示出错信息并停机。如果主设备号不是 5 而是 4，
    // 我们就可以从设备号中取出子设备号。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）。
138         if (MAJOR(dev) == 5) {
139                 dev=current->tty;
140                 if (dev<0)
```

```
141                     panic("tty_ioctl: dev<0");
142             } else
143                     dev=MINOR(dev);
    // 然后根据子设备号和 tty 表，我们可取得对应终端的 tty 结构。于是让 tty 指向对应子设备
    // 号的 tty 结构。然后再根据参数提供的 ioctl 命令 cmd 进行分别处理。144 行后半部分用于
    // 根据子设备号 dev 在 tty_table[]表中选择对应的 tty 结构。如果 dev = 0，表示正在使用
    // 前台终端，因此直接使用终端号 fg_console 作为 tty_table[] 项索引取 tty 结构。如果
    // dev 大于 0，那么就要分两种情况考虑：① dev 是虚拟终端号；② dev 是串行终端号或者
    // 伪终端号。对于虚拟终端其 tty 结构在 tty_table[]中索引项是 dev-1（0 -- 63）。对于
    // 其它类型终端，则它们的 tty 结构索引项就是 dev。例如，如果 dev = 64，表示是一个串
    // 行终端 1，则其 tty 结构就是 ttb_table[dev]。 如果 dev = 1，则对应终端的 tty 结构是
    // tty_table[0]。参见 tty_io.c 程序第 70 -- 73 行。
144             tty = tty_table + (dev ? ((dev < 64)? dev-1:dev) : fg_console);
145             switch (cmd) {
    // 取相应终端 termios 结构信息。此时参数 arg 是用户缓冲区指针。
146                     case TCGETS:
147                             return get_termios(tty,(struct termios *) arg);
    // 在设置 termios 结构信息之前，需要先等待输出队列中所有数据处理完毕，并且刷新（清空）
    // 输入队列。再接着执行下面设置终端 termios 结构的操作。
148                     case TCSETSF:
149                             flush(tty->read_q); /* fallthrough */ /* 接着继续执行 */
    // 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
    // 参数会影响输出的情况，就需要使用这种形式。
150                     case TCSETSW:
151                             wait_until_sent(tty); /* fallthrough */
    // 设置相应终端 termios 结构信息。此时参数 arg 是保存 termios 结构的用户缓冲区指针。
152                     case TCSETS:
153                             return set_termios(tty,(struct termios *) arg, dev);
    // 取相应终端 termio 结构中的信息。此时参数 arg 是用户缓冲区指针。
154                     case TCGETA:
155                             return get_termio(tty,(struct termio *) arg);
    // 在设置 termio 结构信息之前，需要先等待输出队列中所有数据处理完毕，并且刷新（清空）
    // 输入队列。再接着执行下面设置终端 termio 结构的操作。
156                     case TCSETAF:
157                             flush(tty->read_q); /* fallthrough */ /* 接着继续执行 */
    // 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
    // 参数会影响输出的情况，就需要使用这种形式。
158                     case TCSETAW:
159                             wait_until_sent(tty); /* fallthrough */
    // 设置相应终端 termio 结构信息。此时参数 arg 是保存 termio 结构的用户缓冲区指针。
160                     case TCSETA:
161                             return set_termio(tty,(struct termio *) arg, dev);
    // 如果参数 arg 值是 0，则等待输出队列处理完毕（空），并发送一个 break。
162                     case TCSBRK:
163                             if (!arg) {
164                                     wait_until_sent(tty);
165                                     send_break(tty);
166                             }
167                             return 0;
    // 开始/停止流控制。如果参数 arg 是 TCOOFF（Terminal Control Output OFF），则挂起输出；
    // 如果是 TCOON，则恢复挂起的输出。在挂起或恢复输出同时需要把写队列中的字符输出，以
    // 加快用户交互响应速度。如果 arg 是 TCIOFF（Terminal Control Input ON），则挂起输入；
    // 如果是 TCION，则重新开启挂起的输入。
```

```
168                 case TCXONC:
169                         switch (arg) {
170                         case TCOOFF:
171                                 tty->stopped = 1;       // 停止终端输出。
172                                 tty->write(tty);        // 写缓冲队列输出。
173                                 return 0;
174                         case TCOON:
175                                 tty->stopped = 0;       // 恢复终端输出。
176                                 tty->write(tty);
177                                 return 0;
    // 如果参数 arg 是 TCIOFF，表示要求终端停止输入，于是我们往终端写队列中放入 STOP 字符。
    // 当终端收到该字符时就会暂停输入。如果参数是 TCION，表示要发送一个 START 字符，让终
    // 端恢复传输。
    // STOP_CHAR(tty)定义为 ((tty)->termios.c_cc[VSTOP])，即取终端 termios 结构控制字符数
    // 组对应项值。若内核定义了_POSIX_VDISABLE（\0），那么当某一项值等于_POSIX_VDISABLE
    // 的值时，表示禁止使用相应的特殊字符。因此这里直接判断该值是否为 0 来确定要不要把停
    // 止控制字符放入终端写队列中。以下同。
178                         case TCIOFF:
179                                 if (STOP_CHAR(tty))
180                                         PUTCH(STOP_CHAR(tty),tty->write_q);
181                                 return 0;
182                         case TCION:
183                                 if (START_CHAR(tty))
184                                         PUTCH(START_CHAR(tty),tty->write_q);
185                                 return 0;
186                         }
187                         return -EINVAL; /* not implemented */
    // 刷新已写输出但还没有发送、或已收但还没有读的数据。如果参数 arg 是 0，则刷新（清空）
    // 输入队列；如果是 1，则刷新输出队列；如果是 2，则刷新输入和输出队列。
188                 case TCFLSH:
189                         if (arg==0)
190                                 flush(tty->read_q);
191                         else if (arg==1)
192                                 flush(tty->write_q);
193                         else if (arg==2) {
194                                 flush(tty->read_q);
195                                 flush(tty->write_q);
196                         } else
197                                 return -EINVAL;
198                         return 0;
    // 设置终端串行线路专用模式。
199                 case TIOCEXCL:
200                         return -EINVAL; /* not implemented */   /* 未实现 */
    // 复位终端串行线路专用模式。
201                 case TIOCNXCL:
202                         return -EINVAL; /* not implemented */
    // 设置 tty 为控制终端。（TIOCNOTTY - 不要控制终端）。
203                 case TIOCSCTTY:
204                         return -EINVAL; /* set controlling term NI */  /* 未实现 */
    // 读取终端进程组号（即读取前台进程组号）。 首先验证用户缓冲区长度，然后复制终端 tty
    // 的 pgrp 字段到用户缓冲区。此时参数 arg 是用户缓冲区指针。
205                 case TIOCGPGRP:                         // 实现库函数 tcgetpgrp()。
206                         verify_area((void *) arg,4);
```

```
207                         put_fs_long(tty->pgrp,(unsigned long *) arg);
208                         return 0;
    // 设置终端进程组号 pgrp（即设置前台进程组号）。 此时参数 arg 是用户缓冲区中进程组号
    // pgrp 的指针。执行该命令的前提条件是进程必须有控制终端。 如果当前进程没有控制终端，
    // 或者 dev 不是其控制终端，或者控制终端现在的确是正在处理的终端 dev，但进程的会话号
    // 与该终端 dev 的会话号不同，则返回无终端错误信息。
209                 case TIOCSPGRP:                         // 实现库函数 tcsetpgrp()。
210                         if ((current->tty < 0) ||
211                             (current->tty != dev) ||
212                             (tty->session != current->session))
213                                 return -ENOTTY;
    // 然后我们就从用户缓冲区中取得欲设置的进程组号，并对该组号的有效性进行验证。如果组
    // 号 pgrp 小于 0，则返回无效组号错误信息；如果 pgrp 的会话号与当前进程的不同，则返回
    // 许可错误信息。否则我们可以设中终端的进程组号为 prgp。此时 prgp 成为前台进程组。
214                         pgrp=get_fs_long((unsigned long *) arg);
215                         if (pgrp < 0)
216                                 return -EINVAL;
217                         if (session_of_pgrp(pgrp) != current->session)
218                                 return -EPERM;
219                         tty->pgrp = pgrp;
220                         return 0;
    // 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。
    // 此时参数 arg 是用户缓冲区指针。
221                 case TIOCOUTQ:
222                         verify_area((void *) arg,4);
223                         put_fs_long(CHARS(tty->write_q),(unsigned long *) arg);
224                         return 0;
    // 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。
    // 此时参数 arg 是用户缓冲区指针。
225                 case TIOCINQ:
226                         verify_area((void *) arg,4);
227                         put_fs_long(CHARS(tty->secondary),
228                                 (unsigned long *) arg);
229                         return 0;
    // 模拟终端输入操作。该命令以一个指向字符的指针作为参数，并假设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。
230                 case TIOCSTI:
231                         return -EINVAL; /* not implemented */   /* 未实现 */
    // 读取终端设备窗口大小信息（参见 termios.h 中的 winsize 结构）。
232                 case TIOCGWINSZ:
233                         return -EINVAL; /* not implemented */
    // 设置终端设备窗口大小信息（参见 winsize 结构）。
234                 case TIOCSWINSZ:
235                         return -EINVAL; /* not implemented */
    // 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）。
236                 case TIOCMGET:
237                         return -EINVAL; /* not implemented */
    // 设置单个 modem 状态控制引线的状态（true 或 false）。
238                 case TIOCMBIS:
239                         return -EINVAL; /* not implemented */
    // 复位单个 MODEM 状态控制引线的状态。
240                 case TIOCMBIC:
241                         return -EINVAL; /* not implemented */
```

```
    // 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。
242                 case TIOCMSET:
243                         return -EINVAL; /* not implemented */
    // 读取软件载波检测标志（1 - 开启；0 - 关闭）。
244                 case TIOCGSOFTCAR:
245                         return -EINVAL; /* not implemented */
    // 设置软件载波检测标志（1 - 开启；0 - 关闭）。
246                 case TIOCSSOFTCAR:
247                         return -EINVAL; /* not implemented */
248                 default:
249                         return -EINVAL;
250         }
251 }
252
```