Manejador de Terminal

tty.c

Joaquín Caraballo Moreno

Manuel Alejandro Díaz Sánchez
© Universidad de las Palmas de Gran Canaria

ÍNDICE

1. CONCEPTOS BÁSICOS....4

1.1 HARDWARE DEL TERMINAL....4

1.1.1 Terminales conectadas vía RS-232....4

1.1.2 Terminales delineadas en memoria (consolas)....5

1.1.3 Terminales X....6

2. MODOS DE OPERACIÓN....7

2.1 ENTRADA DE CARACTERES....7

2.2 SALIDA DE CARACTERES....8

2.3 E/S ASÍNCRONA....8

2.4 MANEJADOR DE TERMINAL (*TTY MANAGER*)....8

2.4.1 Entrada desde el Terminal de tipo Consola....9

2.4.2 Salida al Terminal de tipo Consola....11

3. ESTRUCTURAS PRINCIPALES (TTY.H)....12

3.1 ESTRUCTURAS DEFINIDAS EN TTY.H....12

3.1.1 Sección de entrada....12

3.1.2 Sección de salida....13

3.1.3 Estado y parámetros del terminal....13

3.1.4 Estado de las peticiones de E/S en curso....13

3.1.5 Cosas sueltas....14

3.1.6 tty.h....15

3.2 OTRAS ESTRUCTURAS DE DATOS....17

3.2.1 Selección de los parámetros de operación....17

3.2.2 Variable global tty_timeout....19

4. PROGRAMA FUENTE (TTY.C)....20

4.1 TTY_TASK....21

4.2 DO_READ....23

4.3 DO_WRITE....25

4.4 DO_IOCTL....26

4.5 DO_OPEN....30

4.6 DO_CLOSE....31

4.7 DO_CANCEL....31

4.8 HANDLE_EVENTS....32

4.9 IN_TRANSFER....33

4.10 IN_PROCESS....34

4.11 ECHO....38

4.12 RAWECHO....39

4.13 BACK_OVER....39

4.14 REPRINT....40

4.15 OUT_PROCESS....41

4.16 DEV_IOCTL....42

4.17 SETATTR....44

4.18 TTY_REPLY....45

4.19 SIGCHAR....45

4.20 TTY_ICANCEL....46

4.21 TTY_INIT....46

4.22 TTY_WAKEUP....46

4.23 SETTIMER....47

4.24 TTY_DEVNOP....48

4.25 FUNCIONES DE COMPATIBILIDAD....48

4.25.1 compat_getp....48

4.25.2 compat_getc....49

4.25.3 compat_setp....50

4.25.4 compat_setc *51*
4.25.5 tspd2sgspd *52*
4.25.6 sgspd2tspd *53*
4.25.7 do_ioctl_compat *53*

5. CUESTIONES 55

ESQUEMA RESUMEN 56

1. Conceptos Básicos

El manejador tty es un fichero del *kernel* cuya función es la de controlar las terminales que están conectadas al computador; entendiendo por terminales cualquier tipo de dispositivo de entrada y/o salida, similares a una impresora, un teclado o un monitor. Los anteriores dispositivos presentan grandes diferencias y es tarea del manejador tratar de ocultarlas a los programas de aplicación. Por otro lado, todos ellos tienen en común la necesidad de atender la entrada asíncrona (= inesperada) de caracteres, así como efectuar un tratamiento a la entrada y a la salida de los caracteres especiales de control.

El carácter asíncrono de los dispositivos que hemos enumerado los hace radicalmente diferentes de los dispositivos de bloque en cuanto al núcleo del sistema se refiere. Con los dispositivos de bloque, las interrupciones hardware se reciben siempre al final de una transferencia que ha iniciado explícitamente la tarea de E/S, y una transferencia suele durar milisegundos; no llegan mensajes inesperados, y los mensajes que llegan lo hacen en un lapso razonablemente corto de tiempo después de efectuar alguna petición al controlador hardware. Esto no plantea ningún problema para el principio de cita ("*the rendez-vous principle*") que rige el paso de mensajes en Minix, ya que la tarea correspondiente sabe cuándo ha de ponerse a esperar un mensaje. Con los terminales, sin embargo, los caracteres pueden llegar en cualquier momento y pueden hacerlo con mucha rapidez.

Existen tres partes diferenciadas a tratar, que son:

- Hardware del Terminal
- Software del Terminal
- Manejador del Terminal

1.1 Hardware del Terminal

En base a la forma en que el sistema operativo se comunica con las terminales, las podemos agrupar en tres grandes grupos que a su vez se subdividen en otros:

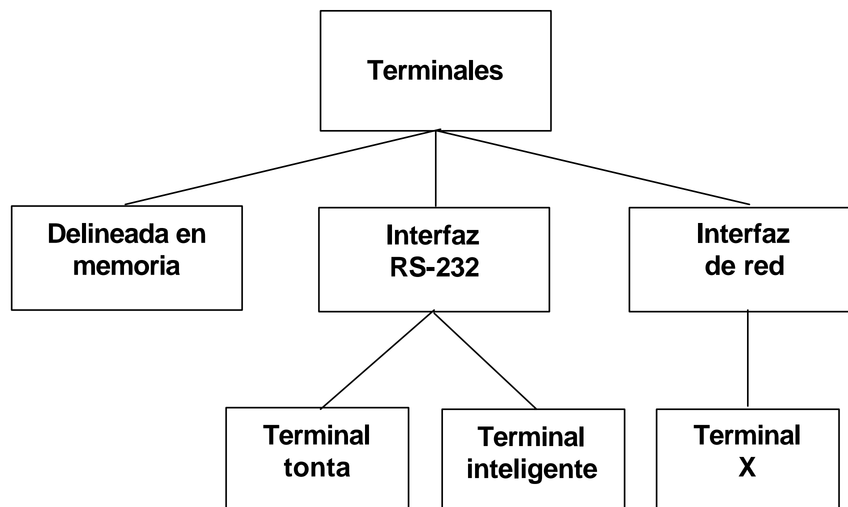

Fig. 1. Tipos de terminales

1.1.1 Terminales conectadas vía RS-232

Normalmente están formadas por un teclado y un monitor y son parte independiente del terminal. Usan un puerto serie para la conexión enviando y recibiendo directamente códigos ASCII adecuadamente interpretados. Es la forma estándar de conexión aunque es lenta al transmitir bit a bit (usa el conector universal RS-232 de 9 ó 25 *pines*) con unas velocidades que van desde los 9600 a los 38400 baudios. Para optimizar su uso suelen acoplar a la tarjeta un chip **UART** (Transmisor Receptor Asíncrono Universal) que se encarga de realizar, en el momento de la transmisión entre el computador y el terminal, las conversiones de bits a caracteres y viceversa. Se suelen utilizar para comunicarse con un computador remoto a través de un módem o una línea telefónica.

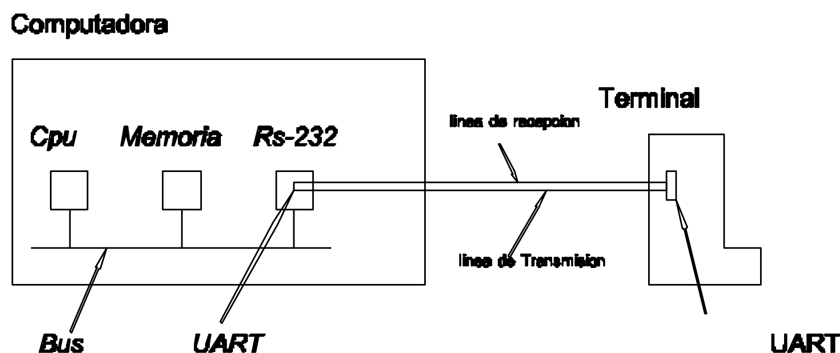

Fig. 2. Terminal conectada a través de la RS-232

Las terminales de copia dura son aquéllas en las que los caracteres tecleados se presentan inmediatamente en la computadora y la salida en impresora. Las de pantalla CRT, denominadas también terminales tontas, tienen una funcionalidad similar pero diferente tecnología. Las terminales CRT inteligentes son pequeñas computadoras con programas complejos que manejan directamente secuencias de escape y otras cosas.

1.1.2 Terminales delineadas en memoria (consolas)

Estas terminales no emplean comunicación serie, siendo parte integral de la máquina; estando sincronizada por una memoria especial llamada **RAM de vídeo** que es parte del espacio de dirección de la computadora y la CPU la direcciona de igual forma que el resto de la memoria. Junto a la RAM de vídeo se encuentra el **controlador de vídeo** que es un chip que extrae los bytes de la memoria de vídeo y genera la señal de vídeo que lee el monitor.

Cada palabra de la memoria puede almacenar caracteres o puntos de pantalla (pixels) según como se quiera mostrar, lo que produce notables diferencias en el modo en que el controlador debe usar esta información. En el primer caso, por ejemplo, podríamos ajustar cada carácter a una caja de 9x14 pixels y dividir la pantalla en 25 líneas x 80 columnas de esas cajitas (para una pantalla de 350x700 pixels).

En el caso del IBM PC, a partir de la dirección 0xB800:0000 se encuentra la RAM de vídeo. Cada carácter está representado por dos bytes, uno de valor y otro de atributo. Para toda la pantalla de 25*80 caracteres se requieren 4000 bytes y viene a ser 174 veces más rápida que una serial.

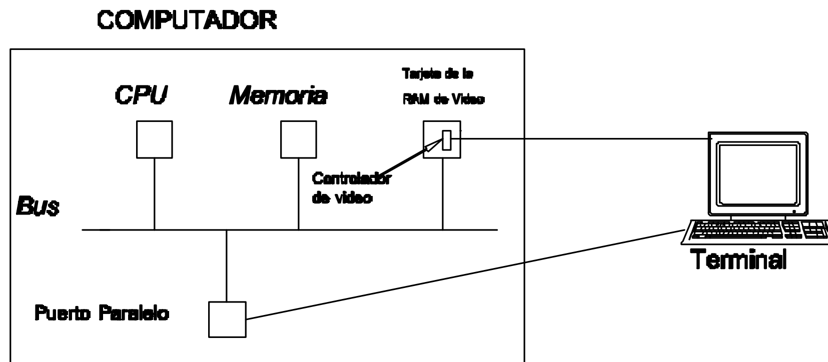

Fig. 3. Terminales Delineadas en Memoria

Cuando el despliegue es por memoria el teclado (totalmente separado de la pantalla) se maneja por un puerto paralelo o serie. Cada vez que se pulsa una tecla se interrumpe la CPU y el manejador de teclado extrae el carácter tecleado leyendo el puerto de E/S. Además es tarea del manejador determinar el estado de teclas como mayúsculas, control, alt…

1.1.3 Terminales X

Son lo último en terminales inteligentes y contienen una CPU tan potente como la del computador principal, con memoria, pantalla de gran resolución, teclado y ratón, y se comunica con éste por medio de una red, como la Ethernet.

Un terminal X es un computador que ejecuta software X. Los programas de un terminal X, llamados servidores X, recogen información del teclado o ratón y aceptan comandos de un computador remoto. Los servidores X se comunican a través de una red con los clientes X, que son ejecutados en algún host remoto.

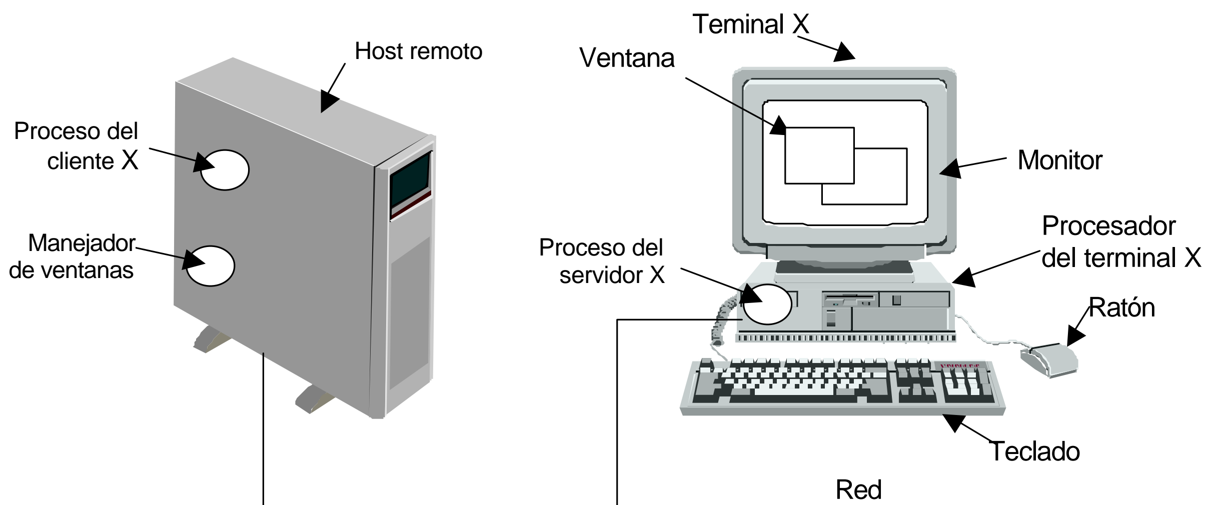

Fig. 4. Cliente y servidor en un sistema de ventanas X

La pantalla de un terminal X contiene un determinado número de ventanas. Cada cliente X es un programa llamado manejador de ventanas, encargado de controlarlas en tareas como la creación, borrado, movimiento, etc.. Para ello, el cliente X envía comando al sevidor X indicándole lo que tiene que hacer.

2. Modos de operación

2.1 Entrada de caracteres

Los dispositivos terminales son full-duplex. Los caracteres de entrada pueden llegar en cualquier momento, incluso cuando el terminal está enviando salida al hardware. Los caracteres entrantes se van almacenando en un buffer. Hay uno de ellos, de tamaño `TTY_IN_BYTES` (= 256, asignado en `tty.h`), para cada dispositivo. Existen dos modalidades de funcionamiento para la entrada de caracteres: la entrada canónica y la entrada no canónica.

En el modo elaborado la entrada se procesa en unidades de línea, delimitadas por caracteres `NL` (nueva línea), `EOL` (fin de línea) o `EOF` (fin de fichero); estos caracteres reciben el nombre genérico de `EOT`. Una operación de lectura no se completará hasta que no haya una línea entera en el buffer de entrada. Además, se devolverá como mucho una línea en cada operación de lectura. Este modo línea a línea permite, mediante la entrada de caracteres especiales, la edición de la línea antes de que el proceso de usuario reciba una copia de la línea completa: la parte del buffer de entrada que no termina en un carácter de final de línea puede editarse usando los caracteres `ERASE` y `KILL`. La entrada del carácter `ERASE` tiene como efecto la eliminación del último carácter introducido antes que él, mientras que la entrada del carácter `KILL` produce la destrucción de toda la línea actual. Esta edición no tiene efecto en la parte del buffer que está protegida por un carácter `EOT` (puede haber varias líneas en el buffer).

En modo elaborado, si llegan más caracteres que los que la cola de entrada puede amortiguar, entonces los nuevos caracteres resultan simplemente descartados.

En el modo de entrada no elaborado la lectura se produce carácter a carácter y por tanto la edición intralínea no es posible. Los caracteres `ERASE` y `KILL` no tienen ningún significado especial. Dos parámetros definidos por el usuario controlan el procesamiento de los caracteres entrantes: `MIN` es el número mínimo de caracteres que tienen que recibirse en el buffer para considerar que una operación de lectura está satisfecha; `TIME` es el valor inicial, en décimas de segundo, de un temporizador *intercarácter* cuya expiración da por terminada la operación de lectura. Ambos parámetros pueden ser nulos, lo cual produce un espectro muy rico de comportamientos frente a la entrada:

- Caso A. `MIN` > 0, `TIME` > 0. Al llegar el primer carácter, el temporizador se inicia. Si se reciben `MIN` caracteres sin que el temporizador expire en la lectura de ninguno de ellos (el temporizador se reinicia cuando llega cada carácter), entonces la lectura se considera satisfecha. Si el temporizador llega a expirar en algún momento, la lectura se interrumpe y los caracteres leídos hasta el momento se devuelven al proceso llamador. Observemos que el proceso llamador puede quedarse colgado para siempre si el primer carácter no llega.
- Caso B. `MIN` > 0, `TIME` = 0. El temporizador no tiene significado y sólo cuenta `MIN`: la lectura no resulta satisfecha hasta que no se han recibido `MIN` caracteres. El proceso llamador puede quedarse colgado para siempre si el número mínimo de caracteres no llega.
- Caso C. `MIN` = 0, `TIME` > 0. En este caso `TIME` no puede ser un temporizador intercarácter ya que `MIN` = 0. `TIME` es simplemente un temporizador que se inicia al efectuarse la llamada. La lectura resulta satisfecha al recibirse un sólo carácter o al expirar el temporizador.
- Caso D. `MIN` = 0, `TIME` = 0. El retorno es inmediato: el número de caracteres que se devuelve es el mínimo de entre los que pidió el proceso llamador y los que hay disponibles en el buffer.

En modo no elaborado, si llegan más caracteres que los que la cola de entrada puede aceptar, la tarea rechaza la entrada y hace que el manejador del hardware (una subrutina contenida en `keyboard.c`, `rs232.c` ó `pty.c`) suspenda la lectura de caracteres del dispositivo hardware.

La satisfacción y el retorno de todas las llamadas de lectura que acabamos de describir está condicionada a la sincronicidad con la que se abrió el dispositivo.

Hay unos cuantos caracteres que tienen, opcionalmente, un tratamiento especial a la entrada, pero su utilización no depende del modo elaborado/no elaborado (ver más abajo la forma de configurar el dispositivo):

- `INTR` genera una señal `SIGINT` que se envía a todos los procesos que usan el terminal.
- `QUIT` genera una señal `SIGQUIT` que se envía a todos los procesos que usan el terminal.
- `LNEXT` hace que el siguiente carácter se interprete de manera literal, saltándose el tratamiento especial de entrada.

- `REPRINT` hace que el dispositivo reenvíe a la salida todos los caracteres de la cola entrada que no han sido utilizados (sólo si el eco está activo, ver más abajo las opciones de configuración).
- `STOP` comando que hace que el terminal interrumpa la salida.
- `START` comando que hace que el terminal reanude la salida, si estaba interrumpida.

Finalmente, a gusto del usuario, se puede efectuar un preproceso de la entrada que consiste en ignorar los caracteres `CR` y convertir los `CR` en `NL` o viceversa.

2.2 Salida de caracteres

Los caracteres que se escriben en el dispositivo sufren un postproceso que también es configurable por el usuario, pero en este caso el tratamiento es más simple: inserción de caracteres `CR` (retorno de carro) y traducción del carácter `TAB` a espacios.
Un carácter se envía al dispositivo hardware tan pronto como los caracteres que le preceden han sido absorbidos por el dispositivo. Si el eco está activado, los caracteres de entrada también se reenvían por la salida, mezclándose con los de salida sin orden ni concierto (para "despejar" la presentación se usa el carácter `REPRINT` que explicamos antes).
A diferencia de la entrada, no hay colas de salida. Observemos que la escritura no se produce de manera inesperada: si un proceso pide una escritura, entonces es que realmente tiene algo que escribir. Esto es completamente diferente de la lectura asíncrona, donde no sabemos cuándo van a llegar los caracteres. Además, un proceso siempre se puede bloquear si la escritura no es posible, mientras que el mundo exterior no puede bloquearse: si no podemos hacer frente a la entrada de caracteres, entonces éstos se perderán sin remedio.
Una escritura se considera satisfecha cuando el dispositivo de salida ha logrado canalizar todos los caracteres de la llamada.
La satisfacción y el retorno de las llamadas de escritura está también condicionada a la sincronicidad con la que se abrió el dispositivo.

2.3 E/S asíncrona

El funcionamiento de la entrada/salida puede variar también dependiendo de si el dispositivo se abre en modo síncrono (por defecto) o en modo asíncrono (usando el flag `O_NONBLOCK` en la llamada a `open`).
En modo síncrono, el proceso llamador resulta bloqueado por el servidor FS cuando la lectura o escritura no pueden ser satisfechas (ver más arriba para los criterios de satisfacción de operaciones).
En modo asíncrono, el proceso llamador *nunca* resulta bloqueado, pase lo que pase. Cuando la operación no puede ser satisfecha o no puede ser satisfecha completamente, la llamada retorna con el código `EAGAIN` en la variable global `errno`.

2.4 Manejador de Terminal (*tty manager*)

El manejador de la terminal está formado por diversos archivos (4 normalmente, o 6 si se soportan RS-232 y pseudo-terminales) y juntos constituyen el más grande de este sistema. Cualquier manejador de dispositivo está basado en la existencia de un único proceso principal en espera de mensajes.

Lo que opinamos que es más importante conocer para entender el funcionamiento del manejador de terminales son las rutinas **send** y **receive** encargadas de enviar y recibir mensajes Minix. Cuando un proceso llama a una de estas funciones se queda bloqueado esperando contestación. En el caso de un manejador de dispositivo unitarea, cuando al recibir un mensaje se queda en ejecución no puede recibir otro mensaje de la rutina de interrupción (de teclado por ejemplo). Por esto el mensaje que envía esta rutina se realiza por medio de **interrupt** que chequea unos indicadores para ver el estado de un proceso y retrasar el mensaje. El contenido de este código se encuentra en "**proc.c**".

Los procesos (programas de usuario y *shell*) que se benefician de las funciones del manejador se comunican con éste, principalmente, a través del *File System* que a su vez lo hace con el manejador mediante mensajes entre los que acepta (cada uno con una estructura particular):

1 Lectura de caracteres desde la terminal
2 Escritura de caracteres en la terminal
3 Fijación del modo de la terminal (IOCTL)
4 Carácter disponible para E/S (del procedimiento de interrupción)
5 Cancelación de la solicitud de lectura anterior (del sistema de archivo cuando ocurra una señal)
6 Abrir dispositivo
7 Cerrar dispositivo

El manejador utiliza para el control de la entrada/salida un vector de estructuras del tipo **tty_struct**, una por terminal. Para la entrada, contiene todos los caracteres que se han tecleado pero que ningún programa ha leído aún, solicitudes para leer caracteres que todavía no se han tecleado y los caracteres de supresión, eliminación, interrupción, retiro, inicio y suspensión. En la salida, contiene los parámetros de solicitudes de escritura que no se han terminado, la posición actual en la pantalla en la RAM de vídeo, el byte de atributo corriente del despliegue e información referente a secuencias de escape que están procesando con regularidad. También contiene diversas variables generales, como el modo de la terminal y el puerto de E/S, si hay alguno, que corresponde a la terminal.

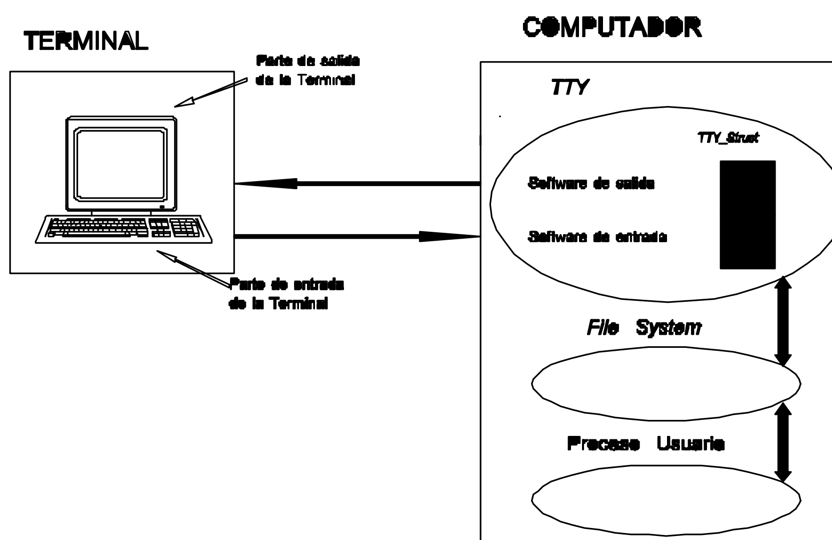

Fig. 5. Esquema de iteraciones del TTY

2.4.1 Entrada desde el Terminal de tipo Consola

Cuando un usuario entra en un sistema con consola, se le crea un shell con /dev/console como entrada, salida y error estándar. El shell comienza tratando de leer de la entrada llamando al procedimiento de biblioteca **read**, que manda un mensaje con el descriptor de fichero, dirección del buffer de entrada, etc., lo que se indica con (1) en la figura. Después de enviar el mensaje, el shell se bloquea esperando respuesta.

El File System recoge el mensaje y localiza el nodo i que corresponde al descriptor anterior. El nodo i es para dev/console y contiene los números de dispositivos mayor y menor (0 para consola y 4 para terminales). El FS busca la tarea del terminal en su mapa de dispositivos con **dmap** y envía un mensaje a la tarea del terminal (2). Normalmente no se ha tecleado nada todavía por lo que no puede satisfacer la solicitud, pero se envía un mensaje al FS desbloqueándolo e indicando que no hay caracteres disponibles (3). El FS registra el hecho de que hay un proceso esperando por entrada y sigue esperando otras peticiones para trabajar. El shell sigue bloqueado esperando.

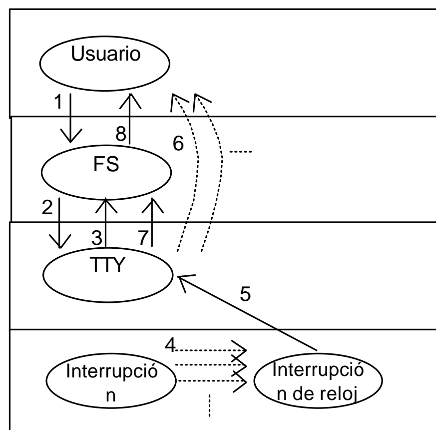

Fig. 6. Petición de lectura desde terminal

Cuando se teclea un carácter se generan 2 interrupciones (una al pulsar y otra al liberar la tecla) y se llama a **scan_keyboard** (lo hace mpx386.s) que extrae el carácter del teclado y lo pone en el buffer de entrada **ibuf** (4). Hay que tener en cuenta que los mensajes de teclas pulsadas no se mandan directamente a la tarea determinada porque esta no está esperando ninguna. En su lugar se activa el **tty_timeout** y se guardan las peticiones en **ibuf**. Cuando se bloquea la tarea de terminal en espera de mensajes (tras n interrupciones) se comprueba el bit anterior y se le envía el mensaje (5) con **tty_wakeup**.

Cuando han llegado suficientes caracteres, el manejador los copia a la dirección dada por el shell (6). Luego manda un mensaje al FS indicándole que ha realizado el trabajo (7) y a su vez, éste le envía un mensaje al shell (8) desbloqueándolo.

Si se han tecleado caracteres antes de que el manejador de terminal esté preparado para ello, entonces **do_read** se encarga de contestar al FS copiando la entrada y desbloqueando el proceso.

A continuación veremos los eventos que ocurren en la tarea de terminal cuando se activa por primera vez con una petición de lectura. Cuando al **tty_task** le llega un mensaje pidiendo caracteres desde teclado, este llama a **do_read** para manejar la petición. A continuación llama a **in_transfer** para obtener cualquier entrada que ya estuviera esperando y luego llama a **handle_events** quien a su vez llama primero a **kb_read** y luego otra vez a **in_transfer** para tratar de extraer del flujo de entrada unos pocos caracteres más. Si la lectura se completa por **handle_events** o por **in_transfer** se manda un mensaje al sistema de ficheros para que desbloquee al llamador.

Ahora, veremos los eventos que ocurren cuando se despierta al terminal por una interrupción de teclado. Cuando se teclea un carácter, este se almacena en el buffer de teclado y se activa un flag para avisar que la consola a experimentado un evento. Entonces, prepara un timer para que finalice en el próximo tic de reloj. La tarea de reloj envía un mensaje a la tarea de terminal indicándole que algo a pasado. Cuando el **tty_task** lo recibe, chequea los flags de eventos de todas las terminales y llama a **handle_events** para cada dispositivo que tenga el flag activado. En el caso del teclado, **handle_events** llamará a **kb_read** y a **in_transfer**, como se hacía con una petición de lectura normal.

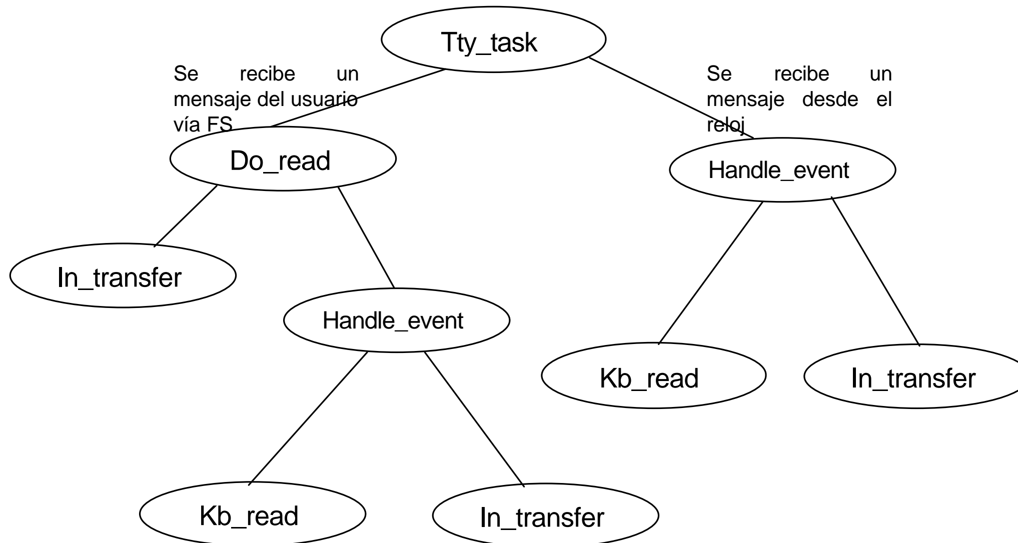

Fig. 7. Manejo de la entrada en el driver de terminal.

2.4.2 Salida al Terminal de tipo Consola

Es algo más sencilla que la entrada puesto que emplea la memoria. Primero el proceso llama a la función **printf** que da formato a la línea. A su vez, esta función llama a **write** que procede al envío de un mensaje al *File System* con un puntero a los caracteres a imprimir, número, etc… Finalmente se envía un mensaje al manejador que recoge los caracteres y los copia procesados en la memoria de vídeo o en el puerto serie.

Cuando llega un mensaje a la tarea del terminal para escribir algo por pantalla se llama al procedimiento **do_write**, que carga los parámetros necesarios en tty_struct. A continuación se llama a **handle_events** para que solape la entrada y la salida, es decir, si se produce alguna entrada recoge el carácter y lo añade al buffer de salida. Luego llama a **cons_write** que copia bloques de caracteres del proceso de usuario al buffer local. Si se van a imprimir caracteres especiales (secuencias de escape), se llamará a **out_char** que sacará los caracteres y además realizará esas otras acciones especiales (saltos de línea, scroll, posicionamiento del cursor, etc.). Los caracteres a imprimir se copian en ramqueue. Finalmente, el procedimiento **flush** copiará datos desde ramqueue a la memoria de vídeo. Todo esto se encuentra contenido en "**console.c**".

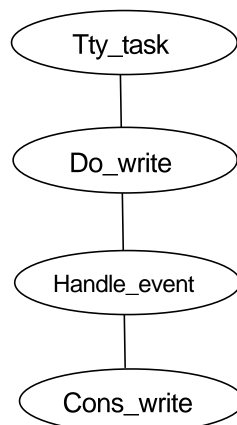

Fig. 8. Manejo de la salida en el driver de terminal.

3. Estructuras principales (tty.h)

3.1 Estructuras definidas en tty.h

Para cada dispositivo terminal manejado hay una entrada en la tabla `tty_table`. Las entradas de esta tabla son instancias de la estructura `tty`; en ella se guarda información sobre el dispositivo: su configuración actual, parámetros de la posible operación que pudiera estar en curso sobre el dispositivo, así como las colas de entrada. He aquí los campos de la estructura.

3.1.1 Sección de entrada

`int tty_events;`

Indicador que avisa que han ocurrido eventos que TTY debería procesar.

`u16_t *tty_inhead;`

Cabecera de la cola circular de entrada (por aquí se insertan caracteres).

`u16_t *tty_intail;`

Último de la cola de entrada (siguiente carácter a enviar al usuario).

`int tty_incount;`

Nº caracteres que hay en la cola de entrada.

`int tty_eotct;`

Nº de EOTs que hay en la cola de entrada (en modo no elaborado todos los caracteres tienen el bit de `EOT` activado, así que este campo cuenta caracteres en este caso).

`devfun_t tty_devread;`

Llamada para lectura al manejador hardware `keyboard.c`, `rs232.c`, ...).

`devfun_t tty_icancel;`

Ordena al manejador hardware que deseche todo lo que ha leído.

`int tty_min;`

Nº mínimo de caracteres requeridos en la cola de entrada para que la lectura sea satisfecha (es el famoso MIN).

`clock_t tty_time;`

Valor del temporizador para este terminal (es el TIME explicado más arriba).

`struct tty *tty_timenext;`

Puntero hacia el siguiente en la lista de terminales que tienen el temporizador activado (ver `tty_timelist` más abajo).

3.1.2 Sección de salida

`devfun_t tty_devwrite;`

Llamada para escritura al manejador hardware (`keyboard.c`, `rs232.c`, ...).

`devfunarg_t tty_echo;`

Llamada al manejador hardware para hacer eco de los caracteres de entrada.

`devfun_t tty_ocancel;`

Ordena al manejador hardware que deseche todo lo que no ha escrito aún.

`devfun_t tty_break;`

Ordena al manejador que envíe un comando `BREAK` al dispositivo.

3.1.3 Estado y parámetros del terminal

`int tty_position;`

Posición actual en la memoria de pantalla para el eco.

`char tty_reprint;`

Flag activado por los manejadores del hardware (`keyboard.c`, `rs232.c` y `console.c`) cuando es necesario reimprimir la salida.

`char tty_escaped;`

Vale 1 si el carácter previo procesado fue `LNEXT`.

`char tty_inhibited;`

Vale 1 si el carácter previo procesado fue `STOP`. Se supone que los manejadores de bajo nivel respetan este flag y detienen la salida de caracteres.

`char tty_pgrp;`

Entrada en la tabla de procesos del último proceso que ha abierto el terminal.

`char tty_openct;`

Nº de veces que se ha abierto el terminal.

3.1.4 Estado de las peticiones de E/S en curso

`char tty_inrepcode;`

Código para el próximo mensaje a FS (`TASK_REPLY` ó `REVIVE`) en respuesta ante una entrada.

`char tty_incaller;`

Entrada en la tabla de procesos del proceso que envió el mensaje de lectura (normalmente FS).

`char tty_inproc;`

Entrada en la tabla de procesos del proceso que quiere leer del terminal.

`vir_bytes tty_in_vir;`

Dirección donde los caracteres de entrada van a almacenarse (en el espacio de direcciones del proceso de usuario).

`int tty_inleft;`

Nº de caracteres que tienen aún que leerse para que la lectura esté satisfecha (es uno de los parámetros a la llamada `read()`).

`int tty_incum;`

Nº de caracteres leídos hasta ahora.

`char tty_outrepcode;`

Código para el próximo mensaje a FS (`TASK_REPLY` ó `REVIVE`) en respuesta ante una salida.

`char tty_outcaller;`

Entrada en la tabla de procesos del proceso que envió el mensaje de escritura (normalmente FS).

`char tty_outproc;`

Entrada en la tabla de procesos del proceso que quiere escribir en el terminal.

`vir_bytes tty_out_vir;`

Dirección de donde los caracteres de salida van a tomarse (en el espacio de direcciones del proceso de usuario).

`int tty_outleft;`

Nº de caracteres que tienen aún que escribirse para que la escritura esté satisfecha (es uno de los parámetros a la llamada `write()`).

`int tty_outcum;`

Nº de caracteres escritos hasta ahora.

`char tty_iocaller;`

Entrada en la tabla de procesos del proceso que envió el mensaje de `ioctl` (normalmente FS).

`char tty_ioproc;`

Entrada en la tabla de procesos del proceso que quiere hacer el `ioctl`.

`int tty_ioreq;`

Código de la petición de `ioctl`.

`vir_bytes tty_iovir;`

Dirección del buffer del `ioctl` (en el espacio de direcciones del proceso de usuario).

3.1.5 Cosas sueltas

`devfun_t tty_ioctl;`

Llamada al manejador hardware para que efectúe el `ioctl` a bajo nivel, si fuera necesario para el dispositivo.

`devfun_t tty_close;`

Llamada al manejador hardware para que cierre el dispositivo (si necesario).

`void *tty_priv;`

Datos que necesita el manejador para sus cosas (no se utiliza en `tty.c`).

`struct termios tty_termios;`

Atributos del terminal (ver más arriba la descripción de la estructura `termios`).

`struct winsize tty_winsize;`

Tamaño de la ventana de la visualización (cuando tal cosa tiene sentido en el dispositivo).

`u16_t tty_inbuf[TTY_IN_BYTES];`

Espacio para el buffer de entrada de caracteres (el apuntado por `tty_inhead` y `tty_intail`.

Observemos que buena parte de los atributos de la estructura `tty` no tienen sentido para dispositivos concretos y otros atributos son utilizados sólamente por los manejadores de bajo nivel. Además de la tabla `tty_table` hay un puntero global:

`tty_t *tty_timelist;`

que apunta a la lista, encadenada por el campo `tty_timenext`, de terminales que tienen activado el temporizador. Esta lista se mantiene ordenada por el campo `tty_time`. El mínimo valor de los temporizadores (es decir, el valor del campo `tty_time` para el primer elemento de esta lista) se guarda en la variable global

`clock_t tty_timeout;`

(en la línea 33 de `./src/kernel/glo.h`), y juega un papel esencial en el mecanismo de entrada de caracteres.

3.1.6 tty.h

```
#define TTY_IN_BYTES     256  /* tty input queue size */
#define TAB_SIZE           8  /* distance between tab stops */
#define TAB_MASK           7  /* mask to compute a tab stop position */

#define ESC             '\33' /* escape */

#define O_NOCTTY       00400  /* from <fcntl.h>, or cc will choke */
#define O_NONBLOCK     04000

typedef _PROTOTYPE( void (*devfun_t), (struct tty *tp) );
typedef _PROTOTYPE( void (*devfunarg_t), (struct tty *tp, int c) );

typedef struct tty {
  int tty_events;       /* set when TTY should inspect this line */

  /* Input queue.  Typed characters are stored here until read by a program. */
  u16_t *tty_inhead;          /* pointer to place where next char goes */
  u16_t *tty_intail;          /* pointer to next char to be given to prog */
  int tty_incount;            /* # chars in the input queue */
```

```
  int tty_eotct;          /* number of "line breaks" in input queue */
  devfun_t tty_devread;         /* routine to read from low level buffers */
  devfun_t tty_icancel;         /* cancel any device input */
  int tty_min;                  /* minimum requested #chars in input queue */
  clock_t tty_time;             /* time when the input is available */
  struct tty *tty_timenext;     /* for a list of ttys with active timers */

  /* Output section. */
  devfun_t tty_devwrite;        /* routine to start actual device output */
  devfunarg_t tty_echo;         /* routine to echo characters input */
  devfun_t tty_ocancel;         /* cancel any ongoing device output */
  devfun_t tty_break;           /* let the device send a break */

  /* Terminal parameters and status. */
  int tty_position;             /* current position on the screen for echoing */
  char tty_reprint;             /* 1 when echoed input messed up, else 0 */
  char tty_escaped;             /* 1 when LNEXT (^V) just seen, else 0 */
  char tty_inhibited;           /* 1 when STOP (^S) just seen (stops output) */
  char tty_pgrp;          /* slot number of controlling process */
  char tty_openct;              /* count of number of opens of this tty */

  /* Information about incomplete I/O requests is stored here. */
  char tty_inrepcode;           /* reply code, TASK_REPLY or REVIVE */
  char tty_incaller;            /* process that made the call (usually FS) */
  char tty_inproc;              /* process that wants to read from tty */
  vir_bytes tty_in_vir;         /* virtual address where data is to go */
  int tty_inleft;         /* how many chars are still needed */
  int tty_incum;          /* # chars input so far */
  char tty_outrepcode;          /* reply code, TASK_REPLY or REVIVE */
  char tty_outcaller;           /* process that made the call (usually FS) */
  char tty_outproc;             /* process that wants to write to tty */
  vir_bytes tty_out_vir;        /* virtual address where data comes from */
  int tty_outleft;              /* # chars yet to be output */
  int tty_outcum;         /* # chars output so far */
  char tty_iocaller;            /* process that made the call (usually FS) */
  char tty_ioproc;              /* process that wants to do an ioctl */
  int tty_ioreq;          /* ioctl request code */
  vir_bytes tty_iovir;          /* virtual address of ioctl buffer */

  /* Miscellaneous. */
  devfun_t tty_ioctl;           /* set line speed, etc. at the device level */
  devfun_t tty_close;           /* tell the device that the tty is closed */
  void *tty_priv;         /* pointer to per device private data */
  struct termios tty_termios; /* terminal attributes */
  struct winsize tty_winsize; /* window size (#lines and #columns) */

  u16_t tty_inbuf[TTY_IN_BYTES];/* tty input buffer */
} tty_t;

EXTERN tty_t tty_table[NR_CONS+NR_RS_LINES+NR_PTYS];

/* Values for the fields. */
#define NOT_ESCAPED          0  /* previous character is not LNEXT (^V) */
#define ESCAPED              1  /* previous character was LNEXT (^V) */
#define RUNNING              0  /* no STOP (^S) has been typed to stop output */
#define STOPPED              1  /* STOP (^S) has been typed to stop output */

/* Fields and flags on characters in the input queue. */
```

```
#define IN_CHAR       0x00FF  /* low 8 bits are the character itself */
#define IN_LEN        0x0F00  /* length of char if it has been echoed */
#define IN_LSHIFT          8  /* length = (c & IN_LEN) >> IN_LSHIFT */
#define IN_EOT        0x1000  /* char is a line break (^D, LF) */
#define IN_EOF        0x2000  /* char is EOF (^D), do not return to user */
#define IN_ESC        0x4000  /* escaped by LNEXT (^V), no interpretation */

/* Times and timeouts. */
#define TIME_NEVER      ((clock_t) -1 < 0 ? (clock_t) LONG_MAX : (clock_t) -1)
#define force_timeout() ((void) (tty_timeout = 0))

EXTERN tty_t *tty_timelist;   /* list of ttys with active timers */

/* Number of elements and limit of a buffer. */
#define buflen(buf)     (sizeof(buf) / sizeof((buf)[0]))
#define bufend(buf)     ((buf) + buflen(buf))
```

3.2 Otras estructuras de datos

3.2.1 Selección de los parámetros de operación

El manejador de dispositivos puede configurarse para habilitar, inhabilitar o modificar gran parte del comportamiento que hemos descrito más arriba. Para ello se usa la estructura `termios` definida en `/usr/include/termios.h`. La estructura tiene un aspecto como éste:

```
struct termios {
tcflag_t c_iflag;          /* tratamiento a la entrada */
tcflag_t c_oflag;          /* tratamiento a la salida */
tcflag_t c_cflag;          /* control hardware */
tcflag_t c_lflag;          /* control local */
speed_t c_ispeed;          /* velocidad de entrada */
speed_t c_ospeed;          /* velocidad de salida */
cc_t c_cc[NCCS];           /* caracteres y parámetros de control */
};
```

No todos los atributos tienen sentido para todos los dispositivos. Es obvio que, por ejemplo, no sirve de nada especificar velocidades con los pseudo-terminales. Veamos un resumen rápido de las posibilidades:

3.2.1.1 c_iflag

Controla el preproceso a la entrada:

- `ICRNL` Convierte `CR` en `NL`.
- `IGNCR` Ignora los `CR`.
- `INLCR` Convierte `NL` en `CR`.
- `IXON` Habilita el control de salida `START`/`STOP`.
- `IXANY` Permite que cualquier carácter de entrada sirva como carácter `START`.
- `ISTRIP` Caracteres de 7 bits (pone el bit alto a cero).

3.2.1.2 c_oflag

Controla el postproceso a la salida:

- `OPOST` Habilita el postproceso a la salida.
- `ONLCR` Inserta `CR` cada vez que se encuentra con `NL`.
- `XTABS` Convierte `TAB` a espacios.

3.2.1.3 c_cflag

Control hardware elemental:

- `CLOCAL` Ignora las líneas de estado del módem.
- `CREAD` Habilita la recepción de caracteres.
- `CSIZE` Nº de bits que tiene un carácter (`CS5, CS6, CS7` ó `CS8` para indicar 5, 6, 7 u 8 bits).
- `CSTOPB` Dos bits de parada (si este flag no está entonces emite un sólo bit de parada).
- `PARENB` Habilita la generación del bit de paridad.
- `PARODD` Hace que se genere paridad impar (en caso contrario se genera paridad par).

3.2.1.4 c_lflag

Miscelánea de controles locales:

- `ECHO` Habilita el eco de los caracteres de entrada: se reenvían a la salida.
- `ECHOE` Si `ICANON` y `ECHO` están habilitados, entonces `ERASE` y `KILL` se interpretan en el eco enviando la sucesión `BACKSPACE-SPACE-BACKSPACE` (varias veces en el caso de `KILL` hasta borrar la línea completa). Si `ECHOE` no está puesto, estos caracteres se reenvían a la salida como lo que quiera que sean.
- `ECHOK` Si `ICANON` y `ECHO` están habilitados, y `ECHOE` no lo está, entonces emite un `NL` después de `KILL` (algunas personas quieren ver la línea que acaban de borrar encima de la que están tecleando).
- `ECHONL` Si `ICANON` está habilitado, envía `NL` a la salida incluso si `ECHO` no está habilitado.
- `ICANON` Habilita la entrada canónica.
- `IEXTEN` Habilita el uso de `LNEXT` y `REPRINT`.
- `ISIG` Habilita el uso de `INTR` y `QUIT`.
- `NOFLSH` Inhabilita la cancelación de la entrada/salida (destrucción de los caracteres ya leídos o por escribir) que se hace normalmente al enviar una señal al proceso llamador.

3.2.1.5 c_ispeed, c_ospeed

Velocidades de entrada y salida, se usan las constantes siguientes, que indican baudios: `B0`, `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, `B38400`, `B57600` y `B115200`.

3.2.1.6 c_cc

El usuario puede definir qué caracteres quiere utilizar para los caracteres especiales que hemos estado detallando. Casi todos ellos, excepto `NL` y `CR` pueden definirse asignándolos a las entradas de este arreglo:

`c_cc[VEOF]`

define `EOF`

`c_cc[VEOL]`

define `EOL`

`c_cc[VERASE]`

define `ERASE`

`c_cc[VINTR]`

define `INTR`

`c_cc[VKILL]`

define `KILL`

`c_cc[VQUIT]`

define `QUIT`

`c_cc[VSTART]`

define `START`

`c_cc[VSTOP]`

define `STOP`

`c_cc[VREPRINT]`

define `REPRINT`

`c_cc[VLNEXT]`

define `LNEXT`

Además, dos entradas de este arreglo se usan para definir los dos parámetros esenciales del modo de entrada no canónica:

`c_cc[VMIN]`

define `MIN`

`c_cc[VTIME]`

define `TIME` (décimas de seg.)

La estructura `termios` se asigna y se lee mediante las funciones `tcsetattr` y `tcgetattr`, que resultan traducidas a las llamadas `ioctl TCSETS`, `TCSETSW`, `TCSETSF`, `TCGETS`. Detallaremos estas llamadas cuando tengamos ocasión de estudiar el código fuente.

3.2.2 Variable global tty_timeout

Como ya hemos comentado anteriormente, también es de vital importancia la variable global tty_timeout definida en glo.h

```
clock_t tty_timeout;
```

4. Programa fuente (tty.c)

```
#include "kernel.h"
#include <termios.h>
#if ENABLE_SRCCOMPAT || ENABLE_BINCOMPAT
#include <sgtty.h>
#endif
#include <sys/ioctl.h>
#include <signal.h>
#include <minix/callnr.h>
#include <minix/com.h>
#if (CHIP == INTEL)
#include <minix/keymap.h>
#endif
#include "tty.h"
#include "proc.h"

/* Address of a tty structure. */
#define tty_addr(line)  (&tty_table[line])

/* First minor numbers for the various classes of TTY devices. */
#define CONS_MINOR        0
#define LOG_MINOR  15
#define RS232_MINOR      16
#define TTYPX_MINOR     128
#define PTYPX_MINOR     192

/* Macros for magic tty types. */
#define isconsole(tp)   ((tp) < tty_addr(NR_CONS))

/* Macros for magic tty structure pointers. */
#define FIRST_TTY tty_addr(0)
#define END_TTY         tty_addr(sizeof(tty_table) / sizeof(tty_table[0]))

/* A device exists if at least its 'devread' function is defined. */
#define tty_active(tp)  ((tp)->tty_devread != NULL)

/* RS232 lines or pseudo terminals can be completely configured out. */
#if NR_RS_LINES == 0
#define rs_init(tp)     ((void) 0)
#endif
#if NR_PTYS == 0
#define pty_init(tp)    ((void) 0)
#define do_pty(tp, mp)  ((void) 0)
#endif

FORWARD _PROTOTYPE( void do_cancel, (tty_t *tp, message *m_ptr)        );
FORWARD _PROTOTYPE( void do_ioctl, (tty_t *tp, message *m_ptr)         );
FORWARD _PROTOTYPE( void do_open, (tty_t *tp, message *m_ptr)          );
FORWARD _PROTOTYPE( void do_close, (tty_t *tp, message *m_ptr)         );
FORWARD _PROTOTYPE( void do_read, (tty_t *tp, message *m_ptr)          );
FORWARD _PROTOTYPE( void do_write, (tty_t *tp, message *m_ptr)         );
FORWARD _PROTOTYPE( void in_transfer, (tty_t *tp)               );
FORWARD _PROTOTYPE( int echo, (tty_t *tp, int ch)               );
FORWARD _PROTOTYPE( void rawecho, (tty_t *tp, int ch)                  );
```

```
FORWARD _PROTOTYPE( int back_over, (tty_t *tp)                    );
FORWARD _PROTOTYPE( void reprint, (tty_t *tp)                     );
FORWARD _PROTOTYPE( void dev_ioctl, (tty_t *tp)                   );
FORWARD _PROTOTYPE( void setattr, (tty_t *tp)                     );
FORWARD _PROTOTYPE( void tty_icancel, (tty_t *tp)                 );
FORWARD _PROTOTYPE( void tty_init, (tty_t *tp)                    );
FORWARD _PROTOTYPE( void settimer, (tty_t *tp, int on)                  );
#if ENABLE_SRCCOMPAT || ENABLE_BINCOMPAT
FORWARD _PROTOTYPE( int compat_getp, (tty_t *tp, struct sgttyb *sg)    );
FORWARD _PROTOTYPE( int compat_getc, (tty_t *tp, struct tchars *sg)    );
FORWARD _PROTOTYPE( int compat_setp, (tty_t *tp, struct sgttyb *sg)    );
FORWARD _PROTOTYPE( int compat_setc, (tty_t *tp, struct tchars *sg)    );
FORWARD _PROTOTYPE( int tspd2sgspd, (speed_t tspd)                );
FORWARD _PROTOTYPE( speed_t sgspd2tspd, (int sgspd)               );
#if ENABLE_BINCOMPAT
FORWARD _PROTOTYPE( void do_ioctl_compat, (tty_t *tp, message *m_ptr)  );
#endif
#endif

/* Default attributes. */
PRIVATE struct termios termios_defaults = {
  TINPUT_DEF, TOUTPUT_DEF, TCTRL_DEF, TLOCAL_DEF, TSPEED_DEF, TSPEED_DEF,
  {
      TEOF_DEF, TEOL_DEF, TERASE_DEF, TINTR_DEF, TKILL_DEF, TMIN_DEF,
      TQUIT_DEF, TTIME_DEF, TSUSP_DEF, TSTART_DEF, TSTOP_DEF,
      TREPRINT_DEF, TLNEXT_DEF, TDISCARD_DEF,
  },
};
PRIVATE struct winsize winsize_defaults; /* = all zeroes */
```

4.1 tty_task

Esta es la tarea TTY propiamente dicha, el punto de entrada principal. Después de llamar a tty_init para inicializar cada terminal, comienza el bucle infinito de recepción de mensajes típico de todas las tareas de E/S. Sin embargo, hay una importante diferencia en el flujo de control: antes de esperar por los mensajes, se invoca primero a la función handle_events para todos los terminales. Esto se hace así para procurar solucionar la asincronía que existente entre los dispositivos de cuyo manejo se ocupa tty.c y la CPU. La tarea TTY trata siempre, por todos los medios, de atender los dispositivos hardware antes de pasar a hacer otra cosa. La función handle_events invoca en última instancia al manejador de bajo nivel (keyboard.c, console.c, pty.c ó rs232.c) el cual manipula él mismo la cola de entrada a través de la función in_process y postprocesa la salida llamando a out_process. Luego veremos en las funciones do_read y do_write otro ejemplo de esta obsesión por atender al hardware.

Después viene el proceso de mensajes. Si se recibe un mensaje del hardware, entonces se trata de una interrupción provocada por la tarea CLOCK a través de la función tty_wakeup, avisando de que el más pequeño de los temporizadores ha expirado, y entonces hay que procesar, al menos, el final de una lectura y posiblemente muchas más cosas, así que se vuelve al bucle que ejecuta de nuevo handle_events para todos los terminales.

Luego se procesan el resto de los mensajes de manera ordinaria. Primero se identifica el terminal requerido en el mensaje (hay que hacer un "mapping" entre los números de los dispositivos instalados en el núcleo y las entradas de la tabla tty_table), y luego se invoca una de las cinco subrutinas que implementan las cinco operaciones elementales con ficheros: do_read, do_write, do_ioctl, do_open y do_close, o a do_cancel, que responde al mensaje CANCEL. Cada subrutina se encarga de enviar su propio mensaje de respuesta si fuese necesario, ya que la elección del mensaje concreto es bastante sutil como veremos.

```
PUBLIC void tty_task()
```

```
{
  message tty_mess;           /* buffer for all incoming messages */
  register tty_t *tp;
  unsigned line;

  /* Initialize the terminal lines. */
  for (tp = FIRST_TTY; tp < END_TTY; tp++) tty_init(tp);

  /* Display the Minix startup banner. */
  printf("Minix %s.%s  Copyright 1997 Prentice-Hall, Inc.\n\n",
                                  OS_RELEASE, OS_VERSION);

#if (CHIP == INTEL)
  /* Real mode, or 16/32-bit protected mode? */
#if _WORD_SIZE == 4
  printf("Executing in 32-bit protected mode\n\n");
#else
  printf("Executing in %s mode\n\n",
      protected_mode ? "16-bit protected" : "real");
#endif
#endif

  while (TRUE) {
      /* Handle any events on any of the ttys. */
      for (tp = FIRST_TTY; tp < END_TTY; tp++) {
            if (tp->tty_events) handle_events(tp);
      }

      receive(ANY, &tty_mess);

      /* A hardware interrupt is an invitation to check for events. */
      if (tty_mess.m_type == HARD_INT) continue;

      /* Check the minor device number. */
      line = tty_mess.TTY_LINE;
      if ((line - CONS_MINOR) < NR_CONS) {
            tp = tty_addr(line - CONS_MINOR);
      } else
      if (line == LOG_MINOR) {
            tp = tty_addr(0);
      } else
      if ((line - RS232_MINOR) < NR_RS_LINES) {
            tp = tty_addr(line - RS232_MINOR + NR_CONS);
      } else
      if ((line - TTYPX_MINOR) < NR_PTYS) {
            tp = tty_addr(line - TTYPX_MINOR + NR_CONS + NR_RS_LINES);
      } else
      if ((line - PTYPX_MINOR) < NR_PTYS) {
            tp = tty_addr(line - PTYPX_MINOR + NR_CONS + NR_RS_LINES);
            do_pty(tp, &tty_mess);
            continue;             /* this is a pty, not a tty */
      } else {
            tp = NULL;
      }

      /* If the device doesn't exist or is not configured return ENXIO. */
      if (tp == NULL || !tty_active(tp)) {
            tty_reply(TASK_REPLY, tty_mess.m_source,
```

```
                                 tty_mess.PROC_NR, ENXIO);
                continue;
        }

        /* Execute the requested function. */
        switch (tty_mess.m_type) {
            case DEV_READ:      do_read(tp, &tty_mess);       break;
            case DEV_WRITE:     do_write(tp, &tty_mess);      break;
            case DEV_IOCTL:     do_ioctl(tp, &tty_mess);      break;
            case DEV_OPEN:      do_open(tp, &tty_mess);       break;
            case DEV_CLOSE:     do_close(tp, &tty_mess);      break;
            case CANCEL:  do_cancel(tp, &tty_mess);    break;
            default:            tty_reply(TASK_REPLY, tty_mess.m_source,
                                      tty_mess.PROC_NR, EINVAL);
        }
    }
}
```

4.2 do_read

Esta función implementa la operación de lectura. En primer lugar verifica los posibles errores que pudiera haber en la petición, como por ejemplo la existencia de otra operación de lectura en proceso en el mismo dispositivo terminal; para ello mira si el campo tty_inleft de la estructura es positivo. Este campo guarda el número de caracteres que quedan por leer de los que se pidieron originalmente en la llamada.

Si las verificaciones tienen éxito, entonces se registran los datos de la operación que ahora se inicia en los campos tty_inrepcode, tty_incaller, tty_inproc, tty_in_vir y tty_inleft. Ni que decir tiene que la operación de lectura no tiene por qué completarse en esta activación de la subrutina. Esto es sólo el comienzo.

Luego se ponen los atributos del modo de entrada no elaborado si éste estuviera seleccionado. Para ello se asigna tty_min según los valores de tty_termios.c_cc[VTIME] y tty_termios.c_cc[VMIN] (ver más arriba los modos de entrada), y se invoca a la función settimer para colocar o retirar el temporizador si hiciese falta.

La lectura trata de satisfacerse con los caracteres que ya están en la cola de entrada llamando a la función in_transfer, la cual copia caracteres de la cola de entrada al buffer proporcionado en el espacio de direcciones del proceso de usuario. Después, siguiendo con la política de atender siempre que sea posible al hardware, se invoca de nuevo a la función handle_events, que llenará de nuevo la cola de entrada, tratará de satisfacer la escritura pendiente si hay alguna, y volverá a tratar de ejecutar una transferencia de entrada si fuese necesario. Observemos que la llamada previa a in_transfer tiene también por objetivo hacer espacio en la cola de entrada para los nuevos caracteres que pudieran estar esperando en el manejador de bajo nivel.

Si la lectura consigue satisfacerse completa (tty_inleft == 0), entonces se regresa de inmediato. En caso contrario, se le manda un mensaje al FS para que bloquee al proceso de usuario (salvo que la apertura haya sido con el flag O_NONBLOCK, en cuyo caso se regresa con el código EAGAIN, como explicamos antes).

```
PRIVATE void do_read(tp, m_ptr)
register tty_t *tp;           /* pointer to tty struct */
message *m_ptr;               /* pointer to message sent to the task */
{
/* A process wants to read from a terminal. */
  int r;

  /* Check if there is already a process hanging in a read, check if the
   * parameters are correct, do I/O.
   */
  if (tp->tty_inleft > 0) {
      r = EIO;
```

```
  } else
  if (m_ptr->COUNT <= 0) {
      r = EINVAL;
  } else
  if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
      r = EFAULT;
  } else {
      /* Copy information from the message to the tty struct. */
      tp->tty_inrepcode = TASK_REPLY;
      tp->tty_incaller = m_ptr->m_source;
      tp->tty_inproc = m_ptr->PROC_NR;
      tp->tty_in_vir = (vir_bytes) m_ptr->ADDRESS;
      tp->tty_inleft = m_ptr->COUNT;

      if (!(tp->tty_termios.c_lflag & ICANON)
                              && tp->tty_termios.c_cc[VTIME] > 0) {
            if (tp->tty_termios.c_cc[VMIN] == 0) {
                  /* MIN & TIME specify a read timer that finishes the
                   * read in TIME/10 seconds if no bytes are available.
                   */
                  lock();
                  settimer(tp, TRUE);
                  tp->tty_min = 1;
                  unlock();
            } else {
                  /* MIN & TIME specify an inter-byte timer that may
                   * have to be cancelled if there are no bytes yet.
                   */
                  if (tp->tty_eotct == 0) {
                        lock();
                        settimer(tp, FALSE);
                        unlock();
                        tp->tty_min = tp->tty_termios.c_cc[VMIN];
                  }
            }
      }

      /* Anything waiting in the input buffer? Clear it out... */
      in_transfer(tp);
      /* ...then go back for more */
      handle_events(tp);
      if (tp->tty_inleft == 0) return;          /* already done */

      /* There were no bytes in the input queue available, so either suspend
       * the caller or break off the read if nonblocking.
       */
      if (m_ptr->TTY_FLAGS & O_NONBLOCK) {
            r = EAGAIN;                   /* cancel the read */
            tp->tty_inleft = tp->tty_incum = 0;
      } else {
            r = SUSPEND;                        /* suspend the caller */
            tp->tty_inrepcode = REVIVE;
      }
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
}
```

4.3 do_write

Esta función es similar a do_read, sólo que como la salida no es tan urgente como la entrada, y no existen colas de salida, no hay tampoco una función equivalente a in_transfer para salida.

En primer lugar se hacen las verificaciones oportunas en cuanto a la disponibilidad del dispositivo leyendo el campo tty_outleft. Luego se instalan en la estructura los parámetros de la escritura que ahora comienza: tty_outrepcode, tty_outcaller, tty_outproc, tty_out_vir y tty_outleft. Después se invoca a handle_events para que ejecute la escritura si es posible. Como antes, si la escritura se pudo satisfacer entonces se regresa inmediatamente. Finalmente, si la escritura no se pudo satisfacer, se decide si hay que bloquear al proceso de usuario o no según el flag O_NONBLOCK.

```
PRIVATE void do_write(tp, m_ptr)
register tty_t *tp;
register message *m_ptr;      /* pointer to message sent to the task */
{
int r;

  /* Check if there is already a process hanging in a write, check if the
   * parameters are correct, do I/O.
   */
  if (tp->tty_outleft > 0) {
      r = EIO;
  } else
  if (m_ptr->COUNT <= 0) {
      r = EINVAL;
  } else
  if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
      r = EFAULT;
  } else {
      /* Copy message parameters to the tty structure. */
      tp->tty_outrepcode = TASK_REPLY;
      tp->tty_outcaller = m_ptr->m_source;
      tp->tty_outproc = m_ptr->PROC_NR;
      tp->tty_out_vir = (vir_bytes) m_ptr->ADDRESS;
      tp->tty_outleft = m_ptr->COUNT;

      /* Try to write. */
      handle_events(tp);
      if (tp->tty_outleft == 0) return;         /* already done */

      /* None or not all the bytes could be written, so either suspend the
       * caller or break off the write if nonblocking.
       */
      if (m_ptr->TTY_FLAGS & O_NONBLOCK) {            /* cancel the write */
            r = tp->tty_outcum > 0 ? tp->tty_outcum : EAGAIN;
            tp->tty_outleft = tp->tty_outcum = 0;
      } else {
            r = SUSPEND;                        /* suspend the caller */
            tp->tty_outrepcode = REVIVE;
      }
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
}
```

4.4 do_ioctl

La función do_ioctl parece complicada, pero eso es porque tiene que procesar muchas peticiones diferentes. En primer lugar, para cada una de las llamadas, calcula el tamaño de lo que se va a transferir. Esto es importante para validar la legitimidad, en términos de protección de memoria, de la transferencia. Luego procesa cada una de las llamadas:

- **TCGETS** Corresponde a la función tcgetattr, que obtiene los parámetros de funcionamiento del terminal (guardados en el campo tty_termios). En este caso simplemente se copia ese campo al espacio proporcionado por el proceso de usuario.
- **TCSETS** Corresponde a la función tcsetattr, con el parámetro de acción a TCSANOW. Esta llamada asigna inmediatamente los parámetros de funcionamiento del terminal (guardados en el campo tty_termios).
- **TCSETSW** Corresponde a la función tcsetattr, con el parámetro de acción a TCSADRAIN (ver termios(3)). Se quieren asignar los parámetros del terminal, al igual que antes, pero ahora se espera a que la salida pendiente se procese (se mira para ello si tty_outleft > 0), enviando un mensaje al FS para que suspenda al proceso de usuario y registrando los datos de la operación de ioctl en la estructura tty. La operación se completará luego en la función dev_ioctl, invocada por handle_events para todos los terminales.
- **TCSETSF** Corresponde a la función tcsetattr, con el parámetro de acción a TCSAFLUSH (ver termios(3)). Es similar a la anterior, pero además se invoca a tty_icancel para desechar los caracteres que se han leído a bajo nivel y los que se hallan en la cola de entrada.
- **TCDRAIN** Corresponde a la función tcdrain, que simplemente espera a que la salida pendiente se procese; esto se hace como en TCSETSW y la operación se completará en la función dev_ioctl.
- **TCSBRK** Corresponde a la función tcsendbreak. Para implementarla se invoca a la función de bajo nivel apuntada por tty_break.
- **TCFLOW** Corresponde a la función tcflow. Esta función se utiliza para enviar caracteres START/STOP a la salida (esto se hace invocando a la función apuntada por tty_echo), y para inhibir el propio dispositivo local (para ello se activa el flag tty_inhibited, que tiene efectos sobre los manejadores de bajo nivel: rs232.c, pty.c y console.c).
- **TCFLSH** Corresponde a la función tcflush, utilizada para desechar los caracteres ya leídos, los todavía no escritos o ambos. Para ello se invoca a la función tty_icancel y a la función apuntada por el campo tty_ocancel.
- **TIOCGPGRP** Corresponde a la función tcgetpgrp, requerida por POSIX, pero que Minix no soporta. Se devuelve un error.
- **TIOCSPGRP** Corresponde a la función tcsetpgrp, requerida por POSIX, pero que Minix no soporta. Se devuelve un error.
- **TIOCGWINSZ** Llamada ioctl para obtener el tamaño de la ventana, guardado en el campo tty_winsize de la estructura tty. Simplemente se copia el campo al espacio de usuario.
- **TIOCSWINSZ** Llamada ioctl para asignar el tamaño de la ventana, guardado en el campo tty_winsize de la estructura tty. Simplemente se copia el campo desde el buffer proporcionado por el usuario.
- **KIOCSMAP** Llamada ioctl para asignar el mapa de teclado, para lo cual se invoca a la función kbd_loadmap (situada en keyboard.c), la cual no tiene demasiado que ver con nuestra visión abstracta de los terminales.
- **TIOCSFON** Llamada ioctl para asigna la fuente con que se dibujan los caracteres en la pantalla, para lo cual se invoca a la función con_loadfont (situada en console.c).

```
PRIVATE void do_ioctl(tp, m_ptr)
register tty_t *tp;
message *m_ptr;                /* pointer to message sent to task */
{
```

```
  int r;
  union {
      int i;
#if ENABLE_SRCCOMPAT
      struct sgttyb sg;
      struct tchars tc;
#endif
  } param;
  phys_bytes user_phys;
  size_t size;

  /* Size of the ioctl parameter. */
  switch (m_ptr->TTY_REQUEST) {
    case TCGETS:        /* Posix tcgetattr function */
    case TCSETS:        /* Posix tcsetattr function, TCSANOW option */
    case TCSETSW:       /* Posix tcsetattr function, TCSADRAIN option */
    case TCSETSF: /* Posix tcsetattr function, TCSAFLUSH option */
        size = sizeof(struct termios);
        break;

    case TCSBRK:        /* Posix tcsendbreak function */
    case TCFLOW:        /* Posix tcflow function */
    case TCFLSH:        /* Posix tcflush function */
    case TIOCGPGRP:     /* Posix tcgetpgrp function */
    case TIOCSPGRP:    /* Posix tcsetpgrp function */
        size = sizeof(int);
        break;

    case TIOCGWINSZ:    /* get window size (not Posix) */
    case TIOCSWINSZ:   /* set window size (not Posix) */
        size = sizeof(struct winsize);
        break;

#if ENABLE_SRCCOMPAT
    case TIOCGETP:      /* BSD-style get terminal properties */
    case TIOCSETP:     /* BSD-style set terminal properties */
      size = sizeof(struct sgttyb);
      break;

    case TIOCGETC:      /* BSD-style get terminal special characters */
    case TIOCSETC:     /* BSD-style get terminal special characters */
      size = sizeof(struct tchars);
      break;
#endif
#if (MACHINE == IBM_PC)
    case KIOCSMAP:     /* load keymap (Minix extension) */
        size = sizeof(keymap_t);
        break;

    case TIOCSFON:     /* load font (Minix extension) */
        size = sizeof(u8_t [8192]);
        break;

#endif
    case TCDRAIN: /* Posix tcdrain function -- no parameter */
    default:           size = 0;
  }
```

```
if (size != 0) {
    user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, size);
    if (user_phys == 0) {
          tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, EFAULT);
          return;
    }
}

r = OK;
switch (m_ptr->TTY_REQUEST) {
  case TCGETS:
    /* Get the termios attributes. */
    phys_copy(vir2phys(&tp->tty_termios), user_phys, (phys_bytes) size);
    break;

  case TCSETSW:
  case TCSETSF:
  case TCDRAIN:
    if (tp->tty_outleft > 0) {
          /* Wait for all ongoing output processing to finish. */
          tp->tty_iocaller = m_ptr->m_source;
          tp->tty_ioproc = m_ptr->PROC_NR;
          tp->tty_ioreq = m_ptr->REQUEST;
          tp->tty_iovir = (vir_bytes) m_ptr->ADDRESS;
          r = SUSPEND;
          break;
    }
    if (m_ptr->TTY_REQUEST == TCDRAIN) break;
    if (m_ptr->TTY_REQUEST == TCSETSF) tty_icancel(tp);
    /*FALL THROUGH*/
  case TCSETS:
    /* Set the termios attributes. */
    phys_copy(user_phys, vir2phys(&tp->tty_termios), (phys_bytes) size);
    setattr(tp);
    break;

  case TCFLSH:
    phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
    switch (param.i) {
        case TCIFLUSH:      tty_icancel(tp);              break;
        case TCOFLUSH:      (*tp->tty_ocancel)(tp);            break;
        case TCIOFLUSH:     tty_icancel(tp); (*tp->tty_ocancel)(tp);break;
        default:            r = EINVAL;
    }
    break;

  case TCFLOW:
    phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
    switch (param.i) {
        case TCOOFF:
        case TCOON:
          tp->tty_inhibited = (param.i == TCOOFF);
          tp->tty_events = 1;
          break;
        case TCIOFF:
          (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTOP]);
          break;
        case TCION:
```

```
            (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTART]);
            break;
          default:
            r = EINVAL;
        }
        break;

      case TCSBRK:
        if (tp->tty_break != NULL) (*tp->tty_break)(tp);
        break;

      case TIOCGWINSZ:
        phys_copy(vir2phys(&tp->tty_winsize), user_phys, (phys_bytes) size);
        break;

      case TIOCSWINSZ:
        phys_copy(user_phys, vir2phys(&tp->tty_winsize), (phys_bytes) size);
        /* SIGWINCH... */
        break;

#if ENABLE_SRCCOMPAT
      case TIOCGETP:
        compat_getp(tp, &param.sg);
        phys_copy(vir2phys(&param.sg), user_phys, (phys_bytes) size);
        break;

      case TIOCSETP:
        phys_copy(user_phys, vir2phys(&param.sg), (phys_bytes) size);
        compat_setp(tp, &param.sg);
        break;

      case TIOCGETC:
        compat_getc(tp, &param.tc);
        phys_copy(vir2phys(&param.tc), user_phys, (phys_bytes) size);
        break;

      case TIOCSETC:
        phys_copy(user_phys, vir2phys(&param.tc), (phys_bytes) size);
        compat_setc(tp, &param.tc);
        break;
#endif

#if (MACHINE == IBM_PC)
      case KIOCSMAP:
        /* Load a new keymap (only /dev/console). */
        if (isconsole(tp)) r = kbd_loadmap(user_phys);
        break;

      case TIOCSFON:
        /* Load a font into an EGA or VGA card (hs@hck.hr) */
        if (isconsole(tp)) r = con_loadfont(user_phys);
        break;
#endif

#if (MACHINE == ATARI)
      case VDU_LOADFONT:
        r = vdu_loadfont(m_ptr);
        break;
```

```
#endif

/* These Posix functions are allowed to fail if _POSIX_JOB_CONTROL is
 * not defined.
 */
    case TIOCGPGRP:
    case TIOCSPGRP:
    default:
#if ENABLE_BINCOMPAT
      do_ioctl_compat(tp, m_ptr);
      return;
#else
      r = ENOTTY;
#endif
  }

  /* Send the reply. */
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
}
```

4.5 do_open

Esta función implementa la llamada al sistema DEV_OPEN de apertura del dispositivo. Cuando un proceso abre un terminal en Minix, ése terminal se transforma en el terminal controlador del proceso (de él recibirá las señales y la E/S). Los procesos no abren normalmente su entrada/salida estándar, sino que la heredan del shell que los puso en marcha, de manera que los procesos pueden agruparse por el terminal controlador que utilizan. La apertura de terminales es un proceso bastante raro de todas formas, con un significado especial, diferente de la apertura de los ficheros ordinarios. Normalmente es el programa getty quien abre el dispositivo terminal y crea así el grupo de procesos que están controlados por ese terminal. Este comportamiento es una simplificación de un protocolo similar, aunque mucho más complejo, que se usa en otros Unixes y que suele denominarse en inglés Job Access Control.

Esta función mira, en primer lugar, que el dispositivo abierto no sea /dev/log, que se usa para diagnósticos y es de sólo escritura. Luego, si no se ha especificado el flag O_NOCTTY en la llamada, se asigna al campo tty_pgrp el proceso que ha abierto el terminal y se incrementa el campo tty_openct, que cuenta el número de procesos que han abierto este terminal. El campo tty_openct se usa únicamente aquí y en do_close.

```
PRIVATE void do_open(tp, m_ptr)
register tty_t *tp;
message *m_ptr;               /* pointer to message sent to task */
{
int r = OK;

  if (m_ptr->TTY_LINE == LOG_MINOR) {
      /* The log device is a write-only diagnostics device. */
      if (m_ptr->COUNT & R_BIT) r = EACCES;
  } else {
      if (!(m_ptr->COUNT & O_NOCTTY)) {
            tp->tty_pgrp = m_ptr->PROC_NR;
            r = 1;
      }
      tp->tty_openct++;
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
```

```
}
```

4.6 do_close

Esta función implementa la llamada al sistema `DEV_CLOSE` de cierre del dispositivo. Si se trata del último proceso que tenía abierto el dispositivo (para ello se examina `tty_openct`), entonces se cierra el terminal a bajo nivel con `tty_close`, se desechan los caracteres ya leídos que hay en la cola de entrada y los que aún no se han escrito usando las funciones `tty_icancel` y `tty_ocancel`, y se ponen los parámetros del terminal a los valores por defecto (guardados en las variables globales `termios_defaults` y `winsize_defaults`) con la función `setattr`.

```
PRIVATE void do_close(tp, m_ptr)
register tty_t *tp;
message *m_ptr;              /* pointer to message sent to task */
{
/* A tty line has been closed.  Clean up the line if it is the last close. */

  if (m_ptr->TTY_LINE != LOG_MINOR && --tp->tty_openct == 0) {
      tp->tty_pgrp = 0;
      tty_icancel(tp);
      (*tp->tty_ocancel)(tp);
      (*tp->tty_close)(tp);
      tp->tty_termios = termios_defaults;
      tp->tty_winsize = winsize_defaults;
      setattr(tp);
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, OK);
}
```

4.7 do_cancel

Esta función responde al mensaje `CANCEL`. Este mensaje es producido por el FS cuando el proceso de usuario está colgado esperando por E/S y alguien le envía una señal. Entonces, la llamada por la que está esperando el proceso tiene que terminar inmediatamente y las colas deben despejarse. Para ello se verifica el tipo de llamada que bloqueó al proceso (entrada, salida, drenaje de caracteres de salida), se invoca a la función adecuada (`tty_icancel`, `tty_ocancel` y se reinicializan a cero los campos de la estructura `tty` que indican que hay una llamada en curso: `tty_inleft`, `tty_incum` para la entrada, `tty_outleft`, `tty_outcum` para la salida, `tty_ioreq` para el `ioctl`.

```
PRIVATE void do_cancel(tp, m_ptr)
register tty_t *tp;
message *m_ptr;              /* pointer to message sent to task */
{
  int proc_nr;
  int mode;

  /* Check the parameters carefully, to avoid cancelling twice. */
  proc_nr = m_ptr->PROC_NR;
  mode = m_ptr->COUNT;
  if ((mode & R_BIT) && tp->tty_inleft != 0 && proc_nr == tp->tty_inproc) {
      /* Process was reading when killed.  Clean up input. */
      tty_icancel(tp);
      tp->tty_inleft = tp->tty_incum = 0;
  }
```

```
  if ((mode & W_BIT) && tp->tty_outleft != 0 && proc_nr == tp->tty_outproc) {
      /* Process was writing when killed.  Clean up output. */
      (*tp->tty_ocancel)(tp);
      tp->tty_outleft = tp->tty_outcum = 0;
  }
  if (tp->tty_ioreq != 0 && proc_nr == tp->tty_ioproc) {
      /* Process was waiting for output to drain. */
      tp->tty_ioreq = 0;
  }
  tp->tty_events = 1;
  tty_reply(TASK_REPLY, m_ptr->m_source, proc_nr, EINTR);
}
```

4.8 handle_events

Esta es la función que pasa el control al manejador de bajo nivel para que atienda el/los dispositivo(s) hardware. Invoca a las funciones apuntadas por `tty_devread` y `tty_devwrite`, y a `tty_ioctl` mientras haya eventos que atender; ésto último se verifica leyendo el flag `tty_events`, colocado por las funciones `do_ioctl`, `do_cancel`, `in_process`, `setattr`, `sigchar`, `tty_wakeup` y por los propios manejadores de bajo nivel. Luego, si hay alguna lectura pendiente de completación, trata de transferir caracteres de la cola de entrada al proceso de usuario invocando a `in_transfer`. Finalmente, si estamos en modo no elaborado y la anterior transferencia pudo satisfacer la lectura, entonces envía un mensaje de respuesta a FS (sólo si `in_transfer` no lo ha hecho ya).

```
do_ioctl, do_cancel, in_process, setattr, sigchar, tty_wakeup

PUBLIC void handle_events(tp)
tty_t *tp;              /* TTY to check for events. */
{
  char *buf;
  unsigned count;

  do {
      tp->tty_events = 0;

      /* Read input and perform input processing. */
      (*tp->tty_devread)(tp);

      /* Perform output processing and write output. */
      (*tp->tty_devwrite)(tp);

      /* Ioctl waiting for some event? */
      if (tp->tty_ioreq != 0) dev_ioctl(tp);
  } while (tp->tty_events);

  /* Transfer characters from the input queue to a waiting process. */
  in_transfer(tp);

  /* Reply if enough bytes are available. */
  if (tp->tty_incum >= tp->tty_min && tp->tty_inleft > 0) {
      tty_reply(tp->tty_inrepcode, tp->tty_incaller, tp->tty_inproc,
                                              tp->tty_incum);
      tp->tty_inleft = tp->tty_incum = 0;
  }
```

```
}
```

4.9 in_transfer

Esta función transfiere caracteres de la cola de entrada al proceso de usuario. Lo hace extrayendo uno a uno los caracteres de la cola de entrada asociada a la terminal, y guardándolos en un pequeño buffer temporal de 64 caracteres que se copia al espacio de usuario cuando está lleno. De paso, va actualizando los campos relevantes en la estructura `tty`; entre ellos los punteros sobre la cola de entrada (`tty_inhead` y `tty_intail`), y especialmente el contador de caracteres que quedan por satisfacer: `tty_inleft`. Tiene cuidado de no ir más allá de un final de línea genérico (`EOT`) si estamos en modo elaborado. Al final envía un mensaje de respuesta a FS si se ha satisfecho una lectura completa.

Esta función resulta invocada por el manejador de bajo nivel, a través de la función `in_process`, si se detecta que la cola de entrada está a punto de desbordarse.

```
PRIVATE void in_transfer(tp)
register tty_t *tp;          /* pointer to terminal to read from */
{
  int ch;
  int count;
  phys_bytes buf_phys, user_base;
  char buf[64], *bp;

  /* Anything to do? */
  if (tp->tty_inleft == 0 || tp->tty_eotct < tp->tty_min) return;

  buf_phys = vir2phys(buf);
  user_base = proc_vir2phys(proc_addr(tp->tty_inproc), 0);
  bp = buf;
  while (tp->tty_inleft > 0 && tp->tty_eotct > 0) {
      ch = *tp->tty_intail;

      if (!(ch & IN_EOF)) {
            /* One character to be delivered to the user. */
            *bp = ch & IN_CHAR;
            tp->tty_inleft--;
            if (++bp == bufend(buf)) {
                  /* Temp buffer full, copy to user space. */
                  phys_copy(buf_phys, user_base + tp->tty_in_vir,
                                    (phys_bytes) buflen(buf));
                  tp->tty_in_vir += buflen(buf);
                  tp->tty_incum += buflen(buf);
                  bp = buf;
            }
      }

      /* Remove the character from the input queue. */
      if (++tp->tty_intail == bufend(tp->tty_inbuf))
            tp->tty_intail = tp->tty_inbuf;
      tp->tty_incount--;
      if (ch & IN_EOT) {
            tp->tty_eotct--;
            /* Don't read past a line break in canonical mode. */
            if (tp->tty_termios.c_lflag & ICANON) tp->tty_inleft = 0;
      }
  }
```

```
    if (bp > buf) {
        /* Leftover characters in the buffer. */
        count = bp - buf;
        phys_copy(buf_phys, user_base + tp->tty_in_vir, (phys_bytes) count);
        tp->tty_in_vir += count;
        tp->tty_incum += count;
    }

    /* Usually reply to the reader, possibly even if incum == 0 (EOF). */
    if (tp->tty_inleft == 0) {
        tty_reply(tp->tty_inrepcode, tp->tty_incaller, tp->tty_inproc,
                                                  tp->tty_incum);
        tp->tty_inleft = tp->tty_incum = 0;
    }
}
```

4.10 in_process

Los caracteres que se han tecleado se procesan, se almacenan y se les produce eco. Devuelve el nº de caracteres procesados. Esta función es invocada por el manejador de bajo nivel cuando tiene necesidad de insertar caracteres en la cola de entrada. Cada carácter del buffer que le pasa el manejador de bajo nivel resulta examinado en busca de los caracteres especiales, según el modo de entrada en que se encuentre el terminal.

Primero se mira si el terminal tiene que retirar el bit 7 del carácter (ver más arriba la estructura `termios` y el flag `ISTRIP`). Luego pregunta por las extensiones comunes (se trata de extensiones al estándar POSIX que tienen todos los Unixes). Una de ellas es `LNEXT`, que produce que se active el flag `tty_escaped` y que el próximo carácter se tome como literal. En la presentación gráfica, durante el tiempo en que el usuario está pensando el siguiente carácter a teclear, se usa `rawecho` para insertar un "^" para indicarle al usuario que está en modo literal; luego un *backspace* borra este indicador temporal. La otra extensión es `REPRINT`, que invoca a la función `reprint` para producir una reimpresión de los caracteres que se han reenviado (eco) a la salida, y que suelen mezclarse aleatoriamente con la salida de verdad.

Más abajo se tratan las conversiones `LF <-> CR`. Después viene el tratamiento que se produce en el modo de entrada elaborado. Se utiliza la función `back_over` para retirar el último carácter de la cola de entrada, en caso de recibir un carácter `ERASE`, o para eliminar la línea completa, en el caso de `KILL`. La función `back_over` devuelve 0 si se ha topado con un carácter `EOT`, 1 si ha conseguido borrar el último carácter, así que se puede iterar la llamada para borrar la última línea completa. En esta parte se procesan también las opciones `ECHOE` y `ECHOK`, tal como se describe más arriba, ayudándose para ello de las funciones `echo` y `rawecho`. Al final de la parte que procesa la entrada canónica se añade el bit de `EOT` a los tres caracteres que se consideran como `EOT`s genéricos: `NL`, `EOL` y `EOF`, los dos últimos redefinibles por el usuario mediante el campo `c_cc` de la estructura `tty_termios`.

El siguiente tratamiento de la entrada es responder a los caracteres `START` y `STOP`. Al recibir un `STOP` se activa el flag `tty_inhibited`, que tiene efectos sobre los manejadores de bajo nivel (`rs232.c`, `pty.c` y `console.c`) y que se supone debería detener la salida de caracteres por el terminal; se coloca también el flag `tty_events`, para que la próxima vez que se ejecute `handle_events` la parada tenga efecto inmediato. La recepción de `START`, o la recepción de cualquier carácter si está activo el flag `XANY`, borra el flag `tty_inhibited` y, presumiblemente, reanudará la salida por el terminal.

Si el flag `ISIG` esta activo, se procesa la llegada de caracteres `INTR` y `QUIT` enviando la correspondiente señal al proceso de usuario. Esto se hace invocando la función `sigchar`.

En este punto se verifica que haya espacio suficiente en la cola de entrada, puesto que estamos a punto de insertar el nuevo carácter en ella, supuesto que el carácter haya pasado los tests anteriores negativamente. Si no hay espacio en la cola de entrada y estamos en modo elaborado, el nuevo carácter se pierde. En modo no elaborado se interrumpe el bucle de inserción de caracteres y se vuelve al llamador (el

manejador de bajo nivel), con la esperanza de que éste sepa entender que no hay espacio en la cola de entrada. La función devuelve el número de caracteres que lograron insertarse en la cola de entrada.

Si estamos en modo no elaborado, se activa el flag de `EOT` del carácter: en este modo, todos los caracteres se consideran `EOT`s, así que el campo `tty_eotct` cuenta caracteres y no líneas. Si `MIN` > 0 y `TIME` > 0 se activa el temporizador intercarácter llamando a la función `settimer` (recordemos que la otra modalidad de temporizador, usado cuando `MIN` = 0 y `TIME` > 0, se activa en la función `do_read` ya descrita). El nuevo carácter (a estas alturas un carácter ordinario o literal) se envía como eco a la salida si el flag `ECHO` está activado. Esto se hace llamando a la función `echo`. Finalmente, el carácter se inserta en la cola de entrada.

Como dijimos antes, si se detecta que la cola de entrada está a punto de desbordarse, se invoca a la función `in_transfer` para ver si podemos descargarla un poco.

```
PUBLIC int in_process(tp, buf, count)
register tty_t *tp;           /* terminal on which character has arrived */
char *buf;              /* buffer with input characters */
int count;              /* number of input characters */
{
  int ch, sig, ct;
  int timeset = FALSE;
  static unsigned char csize_mask[] = { 0x1F, 0x3F, 0x7F, 0xFF };

  for (ct = 0; ct < count; ct++) {
      /* Take one character. */
      ch = *buf++ & BYTE;

      /* Strip to seven bits? */
      if (tp->tty_termios.c_iflag & ISTRIP) ch &= 0x7F;

      /* Input extensions? */
      if (tp->tty_termios.c_lflag & IEXTEN) {

            /* Previous character was a character escape? */
            if (tp->tty_escaped) {
                  tp->tty_escaped = NOT_ESCAPED;
                  ch |= IN_ESC;     /* protect character */
            }

            /* LNEXT (^V) to escape the next character? */
            if (ch == tp->tty_termios.c_cc[VLNEXT]) {
                  tp->tty_escaped = ESCAPED;
                  rawecho(tp, '^');
                  rawecho(tp, '\b');
                  continue;   /* do not store the escape */
            }

            /* REPRINT (^R) to reprint echoed characters? */
            if (ch == tp->tty_termios.c_cc[VREPRINT]) {
                  reprint(tp);
                  continue;
            }
      }

      /* _POSIX_VDISABLE is a normal character value, so better escape it. */
      if (ch == _POSIX_VDISABLE) ch |= IN_ESC;

      /* Map CR to LF, ignore CR, or map LF to CR. */
```

```
if (ch == '\r') {
      if (tp->tty_termios.c_iflag & IGNCR) continue;
      if (tp->tty_termios.c_iflag & ICRNL) ch = '\n';
} else
if (ch == '\n') {
      if (tp->tty_termios.c_iflag & INLCR) ch = '\r';
}

/* Canonical mode? */
if (tp->tty_termios.c_lflag & ICANON) {

      /* Erase processing (rub out of last character). */
      if (ch == tp->tty_termios.c_cc[VERASE]) {
            (void) back_over(tp);
            if (!(tp->tty_termios.c_lflag & ECHOE)) {
                  (void) echo(tp, ch);
            }
            continue;
      }

      /* Kill processing (remove current line). */
      if (ch == tp->tty_termios.c_cc[VKILL]) {
            while (back_over(tp)) {}
            if (!(tp->tty_termios.c_lflag & ECHOE)) {
                  (void) echo(tp, ch);
                  if (tp->tty_termios.c_lflag & ECHOK)
                        rawecho(tp, '\n');
            }
            continue;
      }

      /* EOF (^D) means end-of-file, an invisible "line break". */
      if (ch == tp->tty_termios.c_cc[VEOF]) ch |= IN_EOT | IN_EOF;

      /* The line may be returned to the user after an LF. */
      if (ch == '\n') ch |= IN_EOT;

      /* Same thing with EOL, whatever it may be. */
      if (ch == tp->tty_termios.c_cc[VEOL]) ch |= IN_EOT;
}

/* Start/stop input control? */
if (tp->tty_termios.c_iflag & IXON) {

      /* Output stops on STOP (^S). */
      if (ch == tp->tty_termios.c_cc[VSTOP]) {
            tp->tty_inhibited = STOPPED;
            tp->tty_events = 1;
            continue;
      }

      /* Output restarts on START (^Q) or any character if IXANY. */
      if (tp->tty_inhibited) {
            if (ch == tp->tty_termios.c_cc[VSTART]
                        || (tp->tty_termios.c_iflag & IXANY)) {
                  tp->tty_inhibited = RUNNING;
                  tp->tty_events = 1;
                  if (ch == tp->tty_termios.c_cc[VSTART])
```

```
                            continue;
                    }
            }
        }

        if (tp->tty_termios.c_lflag & ISIG) {
                /* Check for INTR (^?) and QUIT (^\) characters. */
                if (ch == tp->tty_termios.c_cc[VINTR]
                                || ch == tp->tty_termios.c_cc[VQUIT]) {
                        sig = SIGINT;
                        if (ch == tp->tty_termios.c_cc[VQUIT]) sig = SIGQUIT;
                        sigchar(tp, sig);
                        (void) echo(tp, ch);
                        continue;
                }
        }

        /* Is there space in the input buffer? */
        if (tp->tty_incount == buflen(tp->tty_inbuf)) {
                /* No space; discard in canonical mode, keep in raw mode. */
                if (tp->tty_termios.c_lflag & ICANON) continue;
                break;
        }

        if (!(tp->tty_termios.c_lflag & ICANON)) {
                /* In raw mode all characters are "line breaks". */
                ch |= IN_EOT;

                /* Start an inter-byte timer? */
                if (!timeset && tp->tty_termios.c_cc[VMIN] > 0
                                && tp->tty_termios.c_cc[VTIME] > 0) {
                        lock();
                        settimer(tp, TRUE);
                        unlock();
                        timeset = TRUE;
                }
        }

        /* Perform the intricate function of echoing. */
        if (tp->tty_termios.c_lflag & (ECHO|ECHONL)) ch = echo(tp, ch);

        /* Save the character in the input queue. */
        *tp->tty_inhead++ = ch;
        if (tp->tty_inhead == bufend(tp->tty_inbuf))
                tp->tty_inhead = tp->tty_inbuf;
        tp->tty_incount++;
        if (ch & IN_EOT) tp->tty_eotct++;

        /* Try to finish input if the queue threatens to overflow. */
        if (tp->tty_incount == buflen(tp->tty_inbuf)) in_transfer(tp);
  }
  return ct;
}
```

4.11 echo

Esta función se encarga de hacer eco de los caracteres (reenviar a la salida los caracteres de la cola de entrada) si el flag `ECHO` está activo. En primer lugar, si el flag `ECHONL` está activo, reenvía a la salida los caracteres `NL` incluso si el eco está inhabilitado. Luego se procesan los caracteres especiales. En particular, los `TAB`s resultan traducidos a espacios, los `NL` y `CR` se reenvían a la salida tal cual y el resto se precede de "^" y se les asigna una letra mayúscula (simplemente se desplazan en la tabla ASCII). El carácter `DEL` (ASCII 127) se imprime como "^?". Los caracteres ordinarios se reenvían tal cual son. `EOF` se corrige con un `BACKSPACE` justo después de haber sido reenviado (se imprime como "^D" y luego desaparece). Para reenviar los caracteres a la salida se invoca la función apuntada por `tty_echo`, que debería haber sido inicializado para este terminal por el manejador de bajo nivel.

Observemos que un sólo caracter puede reenviarse como varios caracteres. Los cuatro bits inferiores de la parte alta del carácter (que tiene 16 bits) se utilizan para almacenar el número total de caracteres que han resultado de la conversión. Esta cantidad se utilizará luego en la función `back_over` para saber cuántas unidades se debe retroceder en el dispositivo de salida al borrar caracteres de la línea actual.

```
PRIVATE int echo(tp, ch)
register tty_t *tp;           /* terminal on which to echo */
register int ch;        /* pointer to character to echo */
{
  int len, rp;


  ch &= ~IN_LEN;
  if (!(tp->tty_termios.c_lflag & ECHO)) {
      if (ch == ('\n' | IN_EOT) && (tp->tty_termios.c_lflag
                               & (ICANON|ECHONL)) == (ICANON|ECHONL))
            (*tp->tty_echo)(tp, '\n');
      return(ch);
  }

  /* "Reprint" tells if the echo output has been messed up by other output. */
  rp = tp->tty_incount == 0 ? FALSE : tp->tty_reprint;

  if ((ch & IN_CHAR) < ' ') {
      switch (ch & (IN_ESC|IN_EOF|IN_EOT|IN_CHAR)) {
          case '\t':
            len = 0;
            do {
                  (*tp->tty_echo)(tp, ' ');
                  len++;
            } while (len < TAB_SIZE && (tp->tty_position & TAB_MASK) != 0);
            break;
          case '\r' | IN_EOT:
          case '\n' | IN_EOT:
            (*tp->tty_echo)(tp, ch & IN_CHAR);
            len = 0;
            break;
          default:
            (*tp->tty_echo)(tp, '^');
            (*tp->tty_echo)(tp, '@' + (ch & IN_CHAR));
            len = 2;
      }
  } else
  if ((ch & IN_CHAR) == '\177') {
      /* A DEL prints as "^?". */
```

```
        (*tp->tty_echo)(tp, '^');
        (*tp->tty_echo)(tp, '?');
        len = 2;
    } else {
        (*tp->tty_echo)(tp, ch & IN_CHAR);
        len = 1;
    }
    if (ch & IN_EOF) while (len > 0) { (*tp->tty_echo)(tp, '\b'); len--; }

    tp->tty_reprint = rp;
    return(ch | (len << IN_LSHIFT));
}
```

4.12 rawecho

Esta función simplemente hace eco del carácter sin interpretarlo. Si el eco está activado (flag `ECHO`), entonces reenvía por la salida el carácter argumento usando la función apuntada por `tty_echo`.

```
PRIVATE void rawecho(tp, ch)
register tty_t *tp;
int ch;
{
  int rp = tp->tty_reprint;
  if (tp->tty_termios.c_lflag & ECHO) (*tp->tty_echo)(tp, ch);
  tp->tty_reprint = rp;
}
```

4.13 back_over

Esta función se utiliza para borrar un carácter de la línea actual en la cola de entrada. Para ello simplemente hace retroceder el puntero `tty_inhead`. No puede borrar caracteres `EOT`, y cuando se topa con uno no hace nada y devuelve 0 (por lo que en modo no elaborado nunca tendrá efecto). Si tiene éxito devuelve 1. Esto se usa en `in_process` para implementar el efecto de la lectura del carácter `ERASE` e, iterando la llamada, también la del carácter `KILL`. Si el flag `tty_reprint` está activado, es que ha sido colocado por el manejador hardware para indicar que, en la presentación, se han mezclado de manera confusa los caracteres de salida con los del eco. En ese caso se invoca la función `reprint` que reimprime (reenvía a la salida) los caracteres de la última línea de la cola de entrada. Si el flag `ECHOE` está puesto, se utiliza el campo de longitud que se calculó en durante la función `echo` para enviar secuencias `RETROCESO-ESPACIO-RETROCESO` a la salida que permitan actualizar la presentación.

```
PRIVATE int back_over(tp)
register tty_t *tp;
{
  u16_t *head;
  int len;

  if (tp->tty_incount == 0) return(0);    /* queue empty */
  head = tp->tty_inhead;
  if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
  if (*--head & IN_EOT) return(0);        /* can't erase "line breaks" */
  if (tp->tty_reprint) reprint(tp);       /* reprint if messed up */
  tp->tty_inhead = head;
```

```
  tp->tty_incount--;
  if (tp->tty_termios.c_lflag & ECHOE) {
      len = (*head & IN_LEN) >> IN_LSHIFT;
      while (len > 0) {
            rawecho(tp, '\b');
            rawecho(tp, ' ');
            rawecho(tp, '\b');
            len--;
      }
  }
  return(1);                       /* one character erased */
}
```

4.14 reprint

Es posible que los caracteres que se reenvían a la salida como eco se mezclen al azar con la salida ordinaria. Interesa, normalmente, restaurar el contenido del eco ya que representa probablemente algún comando que el usuario está tecleando. La detección de la mezcla de salidas se produce en el manejador de bajo nivel, el cual activa el flag `tty_reprint`. La función `back_over` detecta la activación de este flag e invoca a esta función para reenviar a la salida los caracteres de la última línea de la cola de entrada. También puede provocarse una llamada a reprint si en la entrada aparece un carácter `REPRINT` (`^R`). La función encuentra el comienzo de la línea en la cola de entrada, envía un carácter `REPRINT` seguido de `CR` y `NL` a la salida, y luego manda la línea completa (todo ello utilizando las funciones `rawecho` y `echo`).

```
PRIVATE void reprint(tp)
register tty_t *tp;          /* pointer to tty struct */
{
  int count;
  u16_t *head;

  tp->tty_reprint = FALSE;

  /* Find the last line break in the input. */
  head = tp->tty_inhead;
  count = tp->tty_incount;
  while (count > 0) {
      if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
      if (head[-1] & IN_EOT) break;
      head--;
      count--;
  }
  if (count == tp->tty_incount) return;        /* no reason to reprint */

  /* Show REPRINT (^R) and move to a new line. */
  (void) echo(tp, tp->tty_termios.c_cc[VREPRINT] | IN_ESC);
  rawecho(tp, '\r');
  rawecho(tp, '\n');

  /* Reprint from the last break onwards. */
  do {
      if (head == bufend(tp->tty_inbuf)) head = tp->tty_inbuf;
      *head = echo(tp, *head);
      head++;
      count++;
```

```
  } while (count < tp->tty_incount);
}
```

4.15 out_process

La función `out_process` es la encargada de implementar el postproceso a la salida. Esta función es invocada por los manejadores de bajo nivel y tiene un funcionamiento diferente de `in_process`: no hay cola de salida, así que *no* mueve caracteres; lo que ocurre es que se le pasa un buffer circular ya lleno y la función actúa sobre él, deteniéndose ante la primera modificación que se ve obligada a efectuar, ya que las modificaciones sobreescriben caracteres del buffer. Cuando regresa lo hace con los parámetros de entrada actualizados, de manera que el llamador sabe exactamente dónde debe restaurar el contenido del búfer.

El tratamiento es bastante más simple que el que se produce a la entrada. Las modificaciones consisten en la inserción de `CR`s antes de los `NL`s (si los flags `OPOST` y `ONLCR` están activos) y de la traducción de `TAB`s a espacios (si los flags `OPOST` y `XTABS` han sido habilitados). Al final se actualiza el campo `tty_position`, que indica la posición actual en la presentación, y es utilizado por la función `echo` para la expansión de los `TAB`s.

```
PUBLIC void out_process(tp, bstart, bpos, bend, icount, ocount)
tty_t *tp;
char *bstart, *bpos, *bend;   /* start/pos/end of circular buffer */
int *icount;                  /* # input chars / input chars used */
int *ocount;                  /* max output chars / output chars used */
{
 /* icount is the number of
 * bytes to process, and the number of bytes actually processed on return.
 * *ocount is the space available on input and the space used on output.
 * (Naturally *icount < *ocount.)
 */

  int tablen;
  int ict = *icount;
  int oct = *ocount;
  int pos = tp->tty_position;

  while (ict > 0) {
      switch (*bpos) {
      case '\7':
            break;
      case '\b':
            pos--;
            break;
      case '\r':
            pos = 0;
            break;
      case '\n':
            if ((tp->tty_termios.c_oflag & (OPOST|ONLCR))
                                          == (OPOST|ONLCR)) {
                  /* Map LF to CR+LF if there is space.  Note that the
                   * next character in the buffer is overwritten, so
                   * we stop at this point.
                   */
                  if (oct >= 2) {
                        *bpos = '\r';
                        if (++bpos == bend) bpos = bstart;
```

```
                        *bpos = '\n';
                        pos = 0;
                        ict--;
                        oct -= 2;
                    }
                    goto out_done;    /* no space or buffer got changed */
                }
                break;
        case '\t':
                /* Best guess for the tab length. */
                tablen = TAB_SIZE - (pos & TAB_MASK);

                if ((tp->tty_termios.c_oflag & (OPOST|XTABS))
                                        == (OPOST|XTABS)) {
                    /* Tabs must be expanded. */
                    if (oct >= tablen) {
                        pos += tablen;
                        ict--;
                        oct -= tablen;
                        do {
                            *bpos = ' ';
                            if (++bpos == bend) bpos = bstart;
                        } while (--tablen != 0);
                    }
                    goto out_done;
                }
                /* Tabs are output directly. */
                pos += tablen;
                break;
        default:
                /* Assume any other character prints as one character. */
                pos++;
        }
        if (++bpos == bend) bpos = bstart;
        ict--;
        oct--;
  }
out_done:
  tp->tty_position = pos & TAB_MASK;

  *icount -= ict; /* [io]ct are the number of chars not used */
  *ocount -= oct; /* *[io]count are the number of chars that are used */
}
```

4.16 dev_ioctl

Esta función resulta invocada en `handle_events` para completar aquellas llamadas de `ioctl` que se iniciaron en `do_ioctl` pero que, por tener que esperar a que los caracteres de salida fueran efectivamente enviados al hardware (drenaje de la salida), tuvieron que ser pospuestas. Estas llamadas son `TCSETSW`, `TCSETSF` y `TCDRAIN`. La última llamada es exactamente eso: esperar por el drenaje de los caracteres de salida, así que simplemente se retorna. Para completar las otras hay que copiar sobre la estructura `tty_termios` los parámetros que antes proporcionó el proceso de usuario e invocar a `setattr` para que el cambio tenga efecto. En el caso de `TCSETSF`, además, hay que desechar los caracteres leídos hasta el momento que no han sido copiados al espacio de usuario, si hubiera alguno, llamando a

`tty_icancel`. Finalmente, si se completó realmente alguna llamada, se envía un mensaje `REVIVE` al FS para que desbloquee al proceso de usuario que estaba esperando por la consecución de la llamada.

```
PRIVATE void dev_ioctl(tp)
tty_t *tp;
{
/* The ioctl's TCSETSW, TCSETSF and TCDRAIN wait for output to finish to make
 * sure that an attribute change doesn't affect the processing of current
 * output.  Once output finishes the ioctl is executed as in do_ioctl().
 */
  phys_bytes user_phys;

  if (tp->tty_outleft > 0) return;       /* output not finished */

  if (tp->tty_ioreq != TCDRAIN) {
      if (tp->tty_ioreq == TCSETSF) tty_icancel(tp);
      user_phys = proc_vir2phys(proc_addr(tp->tty_ioproc), tp->tty_iovir);
      phys_copy(user_phys, vir2phys(&tp->tty_termios),
                              (phys_bytes) sizeof(tp->tty_termios));
      setattr(tp);
  }
  tp->tty_ioreq = 0;
  tty_reply(REVIVE, tp->tty_iocaller, tp->tty_ioproc, OK);
}
```

4.17 setattr

Esta función se invoca después de modificar los atributos del terminal almacenados en `tty_termios` para que los cambios tengan efecto sobre los caracteres que están en la cola de entrada. Si hemos vuelto a modo no elaborado, se recorre esta cola convirtiendo todos los caracteres en `EOT`s; se desconecta el temporizador si hubiera alguno en marcha (llamando a `settimer`); se ajustan el valor de `tty_min` según `MIN` y `TIME`. Si el flag `IXON` ha cambiado entonces hay que desinhibir el terminal, para lo cual se inhabilita el flag `tty_inhibited` y se señaliza el cambio en `tty_events`. Si la velocidad se ha puesto a cero hay que enviar una señal `SIGHUP` al proceso de usuario; esto se hace invocando la función `sigchar`. Finalmente, se llama a la función apuntada por `tty_ioctl` para que ejecute los cambios a bajo nivel.

```
PRIVATE void setattr(tp)
tty_t *tp;
{
  u16_t *inp;
  int count;

  if (!(tp->tty_termios.c_lflag & ICANON)) {
      /* Raw mode; put a "line break" on all characters in the input queue.
       * It is undefined what happens to the input queue when ICANON is
       * switched off, a process should use TCSAFLUSH to flush the queue.
       * Keeping the queue to preserve typeahead is the Right Thing, however
       * when a process does use TCSANOW to switch to raw mode.
       */
      count = tp->tty_eotct = tp->tty_incount;
      inp = tp->tty_intail;
      while (count > 0) {
            *inp |= IN_EOT;
            if (++inp == bufend(tp->tty_inbuf)) inp = tp->tty_inbuf;
            --count;
      }
  }

  /* Inspect MIN and TIME. */
  lock();
  settimer(tp, FALSE);
  unlock();
  if (tp->tty_termios.c_lflag & ICANON) {
      /* No MIN & TIME in canonical mode. */
      tp->tty_min = 1;
  } else {
      /* In raw mode MIN is the number of chars wanted, and TIME how long
       * to wait for them.  With interesting exceptions if either is zero.
       */
      tp->tty_min = tp->tty_termios.c_cc[VMIN];
      if (tp->tty_min == 0 && tp->tty_termios.c_cc[VTIME] > 0)
            tp->tty_min = 1;
  }

  if (!(tp->tty_termios.c_iflag & IXON)) {
      /* No start/stop output control, so don't leave output inhibited. */
      tp->tty_inhibited = RUNNING;
      tp->tty_events = 1;
  }
```

```
  /* Setting the output speed to zero hangs up the phone. */
  if (tp->tty_termios.c_ospeed == B0) sigchar(tp, SIGHUP);

  /* Set new line speed, character size, etc at the device level. */
  (*tp->tty_ioctl)(tp);
}
```

4.18 tty_reply

Esta función se usa simplemente para enviar los mensajes de respuesta al FS. Simplemente llena los campos del mensaje con sus argumentos y hace un `send`.

```
PUBLIC void tty_reply(code, replyee, proc_nr, status)
int code;               /* TASK_REPLY or REVIVE */
int replyee;                  /* destination address for the reply */
int proc_nr;                  /* to whom should the reply go? */
int status;             /* reply code */
{
  message tty_mess;

  tty_mess.m_type = code;
  tty_mess.REP_PROC_NR = proc_nr;
  tty_mess.REP_STATUS = status;
  if ((status = send(replyee, &tty_mess)) != OK)
      panic("tty_reply failed, status\n", status);
}
```

4.19 sigchar

Se utiliza para enviar señales `SIGINT`, `SIGQUIT`, `SIGHUP` y `SIGKILL` (ésta última provocada en `keyboard.c`). Para ello se invoca a la función `cause_sig` relacionada con la tarea `SYSTEM` (contenida en `system.c`). A menos que el flag `NOFLSH` esté activo, se cancelan los caracteres de la cola de entrada y se invoca al `tty_ocancel` de bajo nivel para que haga lo mismo con los que aún no se han escrito. Se retira el flag `tty_inhibited` y se señalizan los cambios en `tty_events`.

```
PUBLIC void sigchar(tp, sig)
register tty_t *tp;
int sig;                /* SIGINT, SIGQUIT, SIGKILL or SIGHUP */
{
  if (tp->tty_pgrp != 0) cause_sig(tp->tty_pgrp, sig);

  if (!(tp->tty_termios.c_lflag & NOFLSH)) {
      tp->tty_incount = tp->tty_eotct = 0;      /* kill earlier input */
      tp->tty_intail = tp->tty_inhead;
      (*tp->tty_ocancel)(tp);             /* kill all output */
      tp->tty_inhibited = RUNNING;
      tp->tty_events = 1;
  }
}
```

4.20 tty_icancel

Esta función se utiliza para desechar los caracteres que han sido leídos pero que todavía no han sido transferidos al proceso de usuario. Para ello se vacía la cola de entrada y se invoca a la función de bajo nivel apuntada por `tty_icancel` en la entrada correspondiente a este terminal de la tabla de estructuras `tty` (que debería haber sido inicializada durante la llamada a `tty_init`).

Observemos que, como no hay cola de salida, no hay función `tty_ocancel` sino que el puntero `tty_ocancel` de la estructura `tty` se invoca directamente para el terminal.

```
PRIVATE void tty_icancel(tp)
register tty_t *tp;
{
  tp->tty_incount = tp->tty_eotct = 0;
  tp->tty_intail = tp->tty_inhead;
  (*tp->tty_icancel)(tp);
}
```

4.21 tty_init

Esta función se invoca una sóla vez por cada terminal desde `tty_task`, para inicializar la estructura `tty` y cada terminal en sí. Primero pone valores por defecto a algunos campos que tienen que estar bien definidos. En particular hay que inicializar los punteros sobre la cola circular de entrada, los parámetros de la línea en `tty_termios`, el número mínimo de caracteres a leer (`tty_min`) y, sobre todo, los punteros hacia todas las funciones que implementan la parte dependiente del dispositivo: `tty_icancel`, `tty_ocancel`, `tty_ioctl` y `tty_close`. Todos se ponen apuntando a la función `tty_devnop`, que no hace nada.

Luego se invoca a la función o macro de inicialización privada según el tipo de terminal. Es de esperar que esta función modifique alguno de los punteros a las funciones que mencionamos más arriba.

```
PRIVATE void tty_init(tp)
tty_t *tp;              /* TTY line to initialize. */
{
  tp->tty_intail = tp->tty_inhead = tp->tty_inbuf;
  tp->tty_min = 1;
  tp->tty_termios = termios_defaults;
  tp->tty_icancel = tp->tty_ocancel = tp->tty_ioctl = tp->tty_close =
                                              tty_devnop;
  if (tp < tty_addr(NR_CONS)) {
      scr_init(tp);
  } else
  if (tp < tty_addr(NR_CONS+NR_RS_LINES)) {
      rs_init(tp);
  } else {
      pty_init(tp);
  }
}
```

4.22 tty_wakeup

Esta función es de suma importancia para la entrada de caracteres por el terminal, ya que se encarga de despertar al controlador del terminal cuando tiene lugar algo que debe ser tratado. Esta es la

función invocada por la tarea CLOCK (en clock.c) cuando detecta que el tiempo indicado por tty_timeout ha expirado (variable global declarada en /usr/src/kernel/glo.h). Esta variable se mantiene en la función settimer con el valor del temporizador más pequeño que hay en lista de terminales tty_timelist, y es empleada, además, por algunas funciones del nivel inferior para indicar la ocurrencia de un evento que ha de ser tratado, como la pulsación de una tecla o la llegada de un carácter de una línea RS232.

Primero, la función tty_wakeup recorre la lista encadenada de terminales con temporizador activado en busca de todos los temporizadores que han expirado y los va arrancando de la lista; están todos al comienzo ya que la lista se mantiene ordenada de menor a mayor. Luego actualiza la variable global tty_timeout con el valor del primer temporizador que encuentra que aún no ha expirado (el próximo que vencerá). Finalmente envía un mensaje a la tarea TTY usando la función interrupt del fichero /usr/src/kernel/proc.c. Este mensaje es recibido en la función tty_task como una interrupción hardware, que tiene como efecto que se revise la lista completa de terminales en busca de cosas que atender, invocando repetidamente a handle_events para cada terminal.

```
PUBLIC void tty_wakeup(now)
clock_t now;                        /* current time */
{
  tty_t *tp;

  /* Scan the timerlist for expired timers and compute the next timeout time. */
  tty_timeout = TIME_NEVER;
  while ((tp = tty_timelist) != NULL) {
      if (tp->tty_time > now) {
            tty_timeout = tp->tty_time;   /* this timer is next */
            break;
      }
      tp->tty_min = 0;              /* force read to succeed */
      tp->tty_events = 1;
      tty_timelist = tp->tty_timenext;
  }

  /* Let TTY know there is something afoot. */
  interrupt(TTY);
}
```

4.23 settimer

Inicializa o retira el temporizador para el terminal dado, según el valor del parámetro on.. Esta función es sensible a interrupciones debidas a tty_wakeup(), por lo que se debe llamar dentro de un bloque lock()-unlock().

Comienza retirando el terminal (si está) de la lista encadenada de terminales que tienen temporizadores activos; esta lista está apuntada por la variable tty_timelist y encadenada por el campo tty_timenext de la estructura tty. Si se trataba de retirar el temporizador, entonces simplemente regresa.

Si se trataba de activar el temporizador, actualiza el campo tty_time de la estructura tty para este terminal con el valor de tty_termios.c_cc[VTIME], convenientemente traducido a ticks de reloj, más el tiempo actual obtenido con la función get_uptime (de clock.c). Luego inserta ordenadamente este terminal en la lista encadenada de terminales con temporizador activo y actualiza la variable global tty_timeout con el valor del primer (= más cercano en el tiempo) temporizador.

```
PRIVATE void settimer(tp, on)
tty_t *tp;              /* line to set or unset a timer on */
```

```
int on;                      /* set timer if true, otherwise unset */
{
  tty_t **ptp;

  /* Take tp out of the timerlist if present. */
  for (ptp = &tty_timelist; *ptp != NULL; ptp = &(*ptp)->tty_timenext) {
      if (tp == *ptp) {
            *ptp = tp->tty_timenext;      /* take tp out of the list */
            break;
      }
  }
  if (!on) return;                        /* unsetting it is enough */

  /* Timeout occurs TIME deciseconds from now. */
  tp->tty_time = get_uptime() + tp->tty_termios.c_cc[VTIME] * (HZ/10);

  /* Find a new place in the list. */
  for (ptp = &tty_timelist; *ptp != NULL; ptp = &(*ptp)->tty_timenext) {
      if (tp->tty_time <= (*ptp)->tty_time) break;
  }
  tp->tty_timenext = *ptp;
  *ptp = tp;
  if (tp->tty_time < tty_timeout) tty_timeout = tp->tty_time;
}
```

4.24 tty_devnop

Esta función no hace nada. Se utiliza en `tty_init` para inicializar los punteros `tty_icancel`, `tty_ocancel`, `tty_ioctl` y `tty_close` de la estructura `tty` para cada terminal. Unas veces se sustituirá por la función correspondiente, otras se mantendrá así, ya que hay casos en los que una función del nivel inferior no tiene ninguna acción que ejecutar.

```
PUBLIC void tty_devnop(tp)
tty_t *tp;
{

}
```

4.25 Funciones de compatibilidad

```
#if ENABLE_SRCCOMPAT || ENABLE_BINCOMPAT
```

4.25.1 compat_getp

```
PRIVATE int compat_getp(tp, sg)
tty_t *tp;
struct sgttyb *sg;
{
/* Translate an old TIOCGETP to the termios equivalent. */
  int flgs;

  sg->sg_erase = tp->tty_termios.c_cc[VERASE];
```

```
  sg->sg_kill = tp->tty_termios.c_cc[VKILL];
  sg->sg_ospeed = tspd2sgspd(cfgetospeed(&tp->tty_termios));
  sg->sg_ispeed = tspd2sgspd(cfgetispeed(&tp->tty_termios));

  flgs = 0;

  /* XTABS  - if OPOST and XTABS */
  if ((tp->tty_termios.c_oflag & (OPOST|XTABS)) == (OPOST|XTABS))
      flgs |= 0006000;

  /* BITS5..BITS8  - map directly to CS5..CS8 */
  flgs |= (tp->tty_termios.c_cflag & CSIZE) << (8-2);

  /* EVENP  - if PARENB and not PARODD */
  if ((tp->tty_termios.c_cflag & (PARENB|PARODD)) == PARENB)
      flgs |= 0000200;

  /* ODDP   - if PARENB and PARODD */
  if ((tp->tty_termios.c_cflag & (PARENB|PARODD)) == (PARENB|PARODD))
      flgs |= 0000100;

  /* RAW    - if not ICANON and not ISIG */
  if (!(tp->tty_termios.c_lflag & (ICANON|ISIG)))
      flgs |= 0000040;

  /* CRMOD  - if ICRNL */
  if (tp->tty_termios.c_iflag & ICRNL)
      flgs |= 0000020;

  /* ECHO   - if ECHO */
  if (tp->tty_termios.c_lflag & ECHO)
      flgs |= 0000010;

  /* CBREAK - if not ICANON and ISIG */
  if ((tp->tty_termios.c_lflag & (ICANON|ISIG)) == ISIG)
      flgs |= 0000002;

  sg->sg_flags = flgs;
  return(OK);
}
```

4.25.2 compat_getc

```
PRIVATE int compat_getc(tp, tc)
tty_t *tp;
struct tchars *tc;
{
/* Translate an old TIOCGETC to the termios equivalent. */

  tc->t_intrc = tp->tty_termios.c_cc[VINTR];
  tc->t_quitc = tp->tty_termios.c_cc[VQUIT];
  tc->t_startc = tp->tty_termios.c_cc[VSTART];
  tc->t_stopc = tp->tty_termios.c_cc[VSTOP];
  tc->t_brkc = tp->tty_termios.c_cc[VEOL];
  tc->t_eofc = tp->tty_termios.c_cc[VEOF];
  return(OK);
}
```

4.25.3 compat_setp

```
PRIVATE int compat_setp(tp, sg)
tty_t *tp;
struct sgttyb *sg;
{
/* Translate an old TIOCSETP to the termios equivalent. */
  struct termios termios;
  int flags;

  termios = tp->tty_termios;

  termios.c_cc[VERASE] = sg->sg_erase;
  termios.c_cc[VKILL] = sg->sg_kill;
  cfsetispeed(&termios, sgspd2tspd(sg->sg_ispeed & BYTE));
  cfsetospeed(&termios, sgspd2tspd(sg->sg_ospeed & BYTE));
  flags = sg->sg_flags;

  /* Input flags */

  /* BRKINT - not changed */
  /* ICRNL  - set if CRMOD is set and not RAW */
  /*          (CRMOD also controls output) */
  termios.c_iflag &= ~ICRNL;
  if ((flags & 0000020) && !(flags & 0000040))
      termios.c_iflag |= ICRNL;

  /* IGNBRK - not changed */
  /* IGNCR  - forced off (ignoring cr's is not supported) */
  termios.c_iflag &= ~IGNCR;

  /* IGNPAR - not changed */
  /* INLCR  - forced off (mapping nl's to cr's is not supported) */
  termios.c_iflag &= ~INLCR;

  /* INPCK  - not changed */
  /* ISTRIP - not changed */
  /* IXOFF  - not changed */
  /* IXON   - forced on if not RAW */
  termios.c_iflag &= ~IXON;
  if (!(flags & 0000040))
      termios.c_iflag |= IXON;

  /* PARMRK - not changed */

  /* Output flags */

  /* OPOST  - forced on if not RAW */
  termios.c_oflag &= ~OPOST;
  if (!(flags & 0000040))
      termios.c_oflag |= OPOST;

  /* ONLCR  - forced on if CRMOD */
  termios.c_oflag &= ~ONLCR;
  if (flags & 0000020)
      termios.c_oflag |= ONLCR;
```

```
  /* XTABS  - forced on if XTABS */
  termios.c_oflag &= ~XTABS;
  if (flags & 0006000)
      termios.c_oflag |= XTABS;

  /* CLOCAL - not changed */
  /* CREAD  - forced on (receiver is always enabled) */
  termios.c_cflag |= CREAD;

  /* CSIZE  - CS5-CS8 correspond directly to BITS5-BITS8 */
  termios.c_cflag = (termios.c_cflag & ~CSIZE) | ((flags & 0001400) >> (8-2));

  /* CSTOPB - not changed */
  /* HUPCL  - not changed */
  /* PARENB - set if EVENP or ODDP is set */
  termios.c_cflag &= ~PARENB;
  if (flags & (0000200|0000100))
      termios.c_cflag |= PARENB;

  /* PARODD - set if ODDP is set */
  termios.c_cflag &= ~PARODD;
  if (flags & 0000100)
      termios.c_cflag |= PARODD;

  /* Local flags */

  /* ECHO         - set if ECHO is set */
  termios.c_lflag &= ~ECHO;
  if (flags & 0000010)
      termios.c_lflag |= ECHO;

  /* ECHOE  - not changed */
  /* ECHOK  - not changed */
  /* ECHONL - not changed */
  /* ICANON - set if neither CBREAK nor RAW */
  termios.c_lflag &= ~ICANON;
  if (!(flags & (0000002|0000040)))
      termios.c_lflag |= ICANON;

  /* IEXTEN - set if not RAW */
  /* ISIG   - set if not RAW */
  termios.c_lflag &= ~(IEXTEN|ISIG);
  if (!(flags & 0000040))
      termios.c_lflag |= (IEXTEN|ISIG);

  /* NOFLSH - not changed */
  /* TOSTOP - not changed */

  tp->tty_termios = termios;
  setattr(tp);
  return(OK);
}
```

4.25.4 compat_setc

```
PRIVATE int compat_setc(tp, tc)
```

```
tty_t *tp;
struct tchars *tc;
{
/* Translate an old TIOCSETC to the termios equivalent. */
  struct termios termios;

  termios = tp->tty_termios;

  termios.c_cc[VINTR] = tc->t_intrc;
  termios.c_cc[VQUIT] = tc->t_quitc;
  termios.c_cc[VSTART] = tc->t_startc;
  termios.c_cc[VSTOP] = tc->t_stopc;
  termios.c_cc[VEOL] = tc->t_brkc;
  termios.c_cc[VEOF] = tc->t_eofc;

  tp->tty_termios = termios;
  setattr(tp);
  return(OK);
}


/* Table of termios line speed to sgtty line speed translations.   All termios
 * speeds are present even if sgtty didn't know about them.  (Now it does.)
 */
PRIVATE struct s2s {
  speed_t   tspd;
  u8_t            sgspd;
} ts2sgs[] = {
  { B0,             0 },
  { B50,      50 },
  { B75,      75 },
  { B110,      1 },
  { B134,    134 },
  { B200,      2 },
  { B300,      3 },
  { B600,      6 },
  { B1200,    12 },
  { B1800,    18 },
  { B2400,    24 },
  { B4800,    48 },
  { B9600,    96 },
  { B19200, 192 },
  { B38400, 195 },
  { B57600, 194 },
  { B115200,      193 },
};
```

4.25.5 tspd2sgspd

```
PRIVATE int tspd2sgspd(tspd)
speed_t tspd;
{
/* Translate a termios speed to sgtty speed. */
  struct s2s *s;

  for (s = ts2sgs; s < ts2sgs + sizeof(ts2sgs)/sizeof(ts2sgs[0]); s++) {
      if (s->tspd == tspd) return(s->sgspd);
```

```
  }
  return 96;
}
```

4.25.6 sgspd2tspd

```
PRIVATE speed_t sgspd2tspd(sgspd)
int sgspd;
{
/* Translate a sgtty speed to termios speed. */
  struct s2s *s;

  for (s = ts2sgs; s < ts2sgs + sizeof(ts2sgs)/sizeof(ts2sgs[0]); s++) {
      if (s->sgspd == sgspd) return(s->tspd);
  }
  return B9600;
}


#if ENABLE_BINCOMPAT
```

4.25.7 do_ioctl_compat

```
PRIVATE void do_ioctl_compat(tp, m_ptr)
tty_t *tp;
message *m_ptr;
{
/* Handle the old sgtty ioctl's that packed the sgtty or tchars struct into
 * the Minix message.  Efficient then, troublesome now.
 */
  int minor, proc, func, result, r;
  long flags, erki, spek;
  u8_t erase, kill, intr, quit, xon, xoff, brk, eof, ispeed, ospeed;
  struct sgttyb sg;
  struct tchars tc;
  message reply_mess;

  minor = m_ptr->TTY_LINE;
  proc = m_ptr->PROC_NR;
  func = m_ptr->REQUEST;
  spek = m_ptr->m2_l1;
  flags = m_ptr->m2_l2;

  switch(func)
  {
    case (('t'<<8) | 8):      /* TIOCGETP */
      r = compat_getp(tp, &sg);
      erase = sg.sg_erase;
      kill = sg.sg_kill;
      ispeed = sg.sg_ispeed;
      ospeed = sg.sg_ospeed;
      flags = sg.sg_flags;
      erki = ((long)ospeed<<24) | ((long)ispeed<<16) | ((long)erase<<8) |kill;
      break;
    case (('t'<<8) | 18):     /* TIOCGETC */
      r = compat_getc(tp, &tc);
      intr = tc.t_intrc;
```

```
      quit = tc.t_quitc;
      xon = tc.t_startc;
      xoff = tc.t_stopc;
      brk = tc.t_brkc;
      eof = tc.t_eofc;
      erki = ((long)intr<<24) | ((long)quit<<16) | ((long)xon<<8) | xoff;
      flags = (eof << 8) | brk;
      break;
    case (('t'<<8) | 17):     /* TIOCSETC */
      tc.t_stopc = (spek >> 0) & 0xFF;
      tc.t_startc = (spek >> 8) & 0xFF;
      tc.t_quitc = (spek >> 16) & 0xFF;
      tc.t_intrc = (spek >> 24) & 0xFF;
      tc.t_brkc = (flags >> 0) & 0xFF;
      tc.t_eofc = (flags >> 8) & 0xFF;
      r = compat_setc(tp, &tc);
      break;
    case (('t'<<8) | 9):      /* TIOCSETP */
      sg.sg_erase = (spek >> 8) & 0xFF;
      sg.sg_kill = (spek >> 0) & 0xFF;
      sg.sg_ispeed = (spek >> 16) & 0xFF;
      sg.sg_ospeed = (spek >> 24) & 0xFF;
      sg.sg_flags = flags;
      r = compat_setp(tp, &sg);
      break;
    default:
      r = ENOTTY;
  }
  reply_mess.m_type = TASK_REPLY;
  reply_mess.REP_PROC_NR = m_ptr->PROC_NR;
  reply_mess.REP_STATUS = r;
  reply_mess.m2_l1 = erki;
  reply_mess.m2_l2 = flags;
  send(m_ptr->m_source, &reply_mess);
}
#endif /* ENABLE_BINCOMPAT */
#endif /* ENABLE_SRCCOMPAT || ENABLE_BINCOMPAT */
```

5. Cuestiones

1. Explica cuales son las estructuras de datos principales que utiliza el manejador de tty
2. ¿Cuál es la secuencia de eventos cuando se solicita una lectura? ¿Y una escritura?
3. ¿Cómo se utiliza el buffer principal de entrada?
4. ¿Cómo se procesan las interrupciones?

6. Esquema resumen