LECCIÓN 23: TECLADO EN LINUX

Teclado 1. Introducción – Historia

Las computadoras son muy usadas en este tiempo, y para algunos forman parte de su vida, y al utilizarla usamos el teclado... y surgen varias preguntas en cuanto al teclado, por ejemplo: ¿Quién fue el que ordenó así las letras del alfabeto? ¿Por que no las ordenaron de manera alfabética?.

Normalmente usamos el teclado QWERTY, llamado asi, debido al orden del las letras que tiene la fila superior.

Pero primero recordemos que antes de que surgieran o fueran siquiera inventadas las computadoras y las máquinas de escribir electricas, se utilizaban las mecánicas, que se comenzaron a conocer durante la primera mitad del siglo XIX. Fue en 1872 cuando se lanza la primera máquina de escribir ampliamente conocida, diseñada por Cristopher Latham Sholes en Milwakee, Estados Unidos, con la ayuda de dos amigos inventores.

El artefacto contaba con las teclas ordenadas en orden alfabético, pero surgió un gran problema. Estas máquinas funcionaban mediante martillos con el inverso de las letras grabadas en su cabeza. Al golpear un tipo de papel a través de una cinta con tinta se marcaba la letra. El problema era que el movimiento de las teclas empujado por la presión de los dedos causaba frecuentes choques de las palancas, con lo que las primeras máquinas se atascaban con mucha frecuencia.

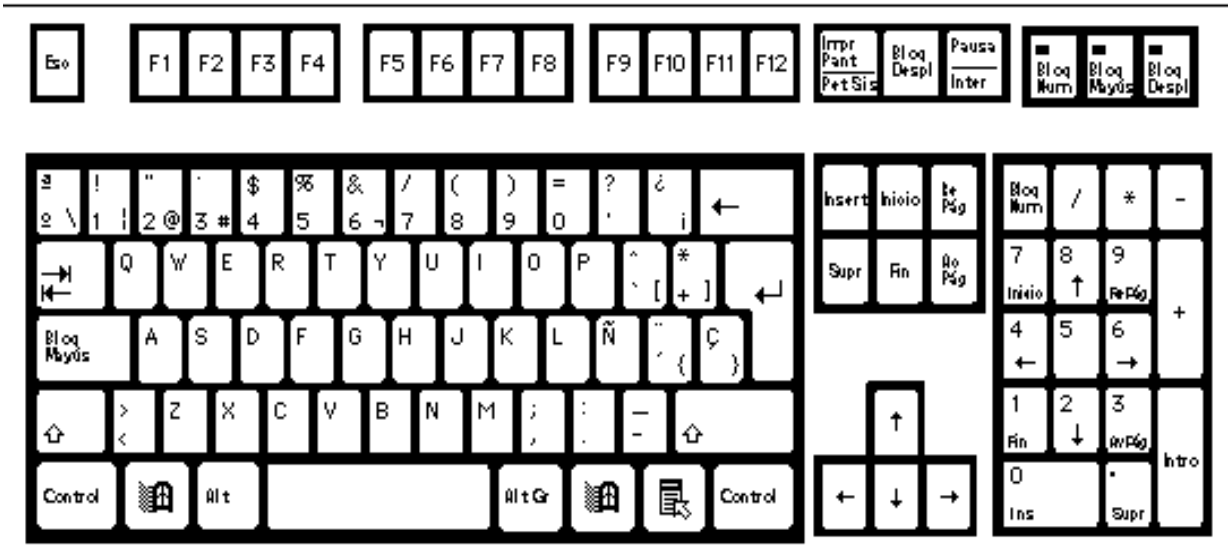
Latham trató de mejorar el diseño de la máquina para eliminar este problema. Para ello, alteró el orden de las teclas con el fin de separar las letras que se usaban juntas con más frecuencia. Para eso hizó un estudio de frecuencia de pares de letras, es decir, los pares que más se utilizaban (en inglés) y que, por consecuencia, causaban la mayoría de los choques. El resultado fue el orden QWERTY, el cual todos conocemos actualmente, aunque no terminó totalmente con el problema, si logró reducirlo.

El teclado que diseñó Cristopher Latham se mantuvo con los modelos que surgieron después, y se difundió por todo el mundo de tal manera, que cuando surgieron las máquinas de escribir eléctricas y luego los teclados para computadoras, el teclado QWERTY continuó utilizandose.

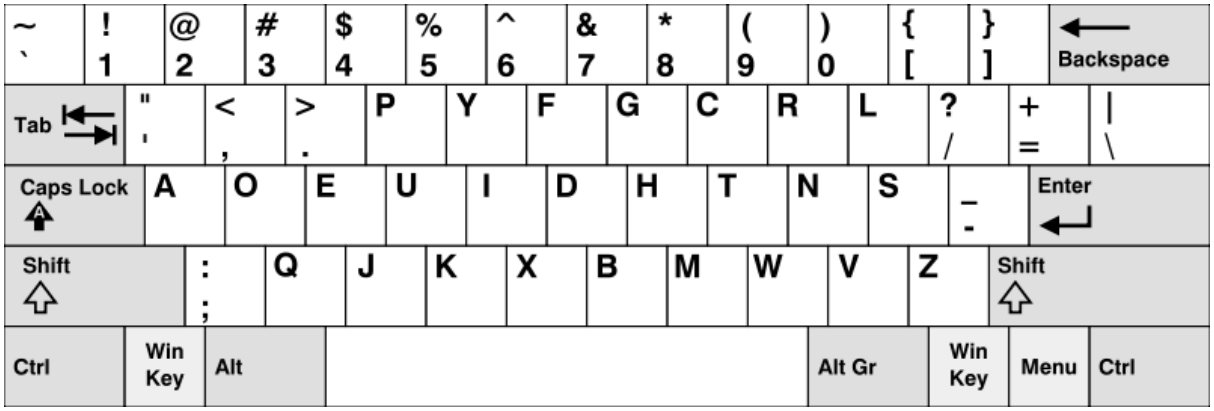
Aunque este orden no es el mejor, es tan popular que se ha convertido en el estándar de facto.

De todos modos, en 1932 un capitán de submarinos e inventor llamado Dvorak diseñó una disposición del teclado que permite escribir más rápidamente. En ese teclado las vocales estan en el centro a la izquierda y las consonantes mas usadas a la derecha. Esto hace que la escritura en ese teclado sea más simple y descansada.

Aunque fue bien recibido por los expertos y se reconocieron las ventajas del teclado Dvorak, la difusión del teclado QWERTY ha hecho casi imposible el cambio.
Teclado QWERTY



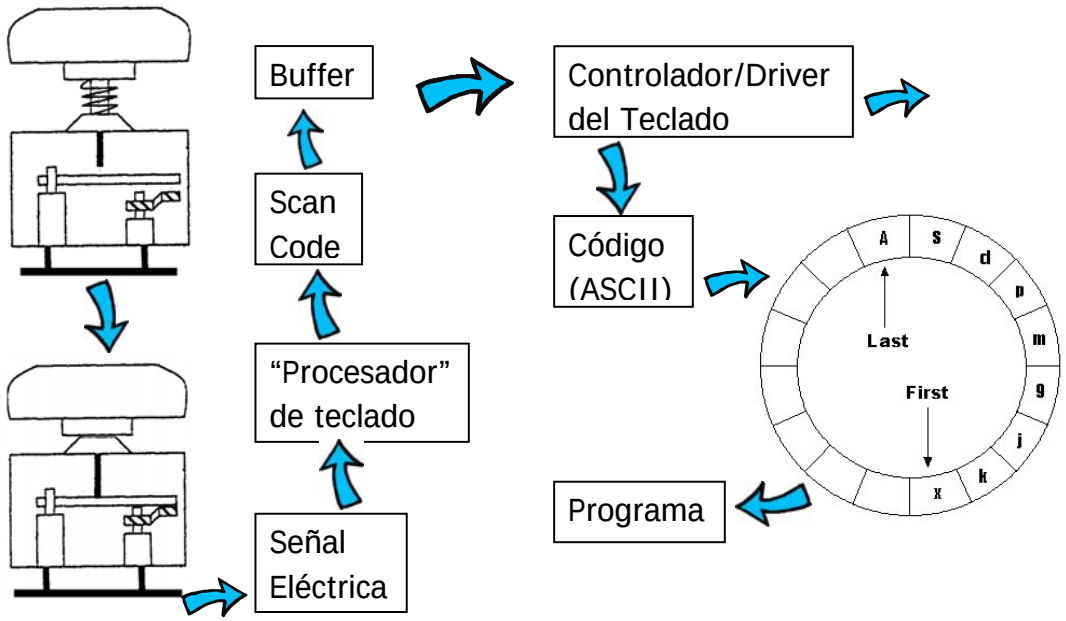
Teclado DVORAK



Teclado 2. Interfaz Teclado - Sistema Operativo

El teclado es el dispositivo más sencillo que puede conectarse al ordenador, además de ser el dispositivo más común de entrada de datos. Un teclado está realizado mediante un microcontrolador. Estos microcontroladores ejecutan sus propios programas que están grabados en sus respectivas ROMs internas. Estos programas realizan la exploración matricial de las teclas para determinar cuales están pulsadas.

Cuando se presiona una tecla, el carácter correspondiente no es simplemente añadido al buffer de entrada de la tty (manejador genérico de terminal), es necesario un procesamiento previo antes de que el kernel sepa cual ha sido el carácter correspondiente a la tecla pulsada.



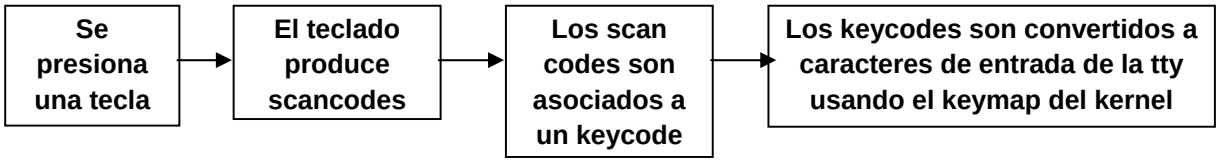
El teclado, consta de una matriz de contactos, que al presionar una tecla, cierran el circuito.

Un microcontrolador detecta la presión de la tecla, y genera un código.

Al soltarse la tecla, se genera otro código. De esta manera el chip localizado en la placa del teclado puede saber cuándo fue presionada y cuándo fue soltada, y actuar en consecuencia.

Los códigos generados son llamados Códigos de barrido (ScanCode).
Una vez detectada la presión de la tecla, los códigos de barrido son generados, y enviados de forma serial a través del cable, y con el conector del teclado llegan a la placa madre del PC.

El código es recibido por el microcontrolador conocido como BIOS DE TECLADO. Este chip compara el código de barrido con el correspondiente a la Tabla de caracteres. Genera una interrupción por hardware, y envía los datos al procesador.



El interfaz de teclado del PC se ocupa de rastrear continuamente el estado de todas las teclas, para detectar si ha ocurrido algún cambio de estado en cualquiera de ellas. De ser así, determina si lo que ha ocurrido es una pulsación o una liberación de tecla, y que tecla ha sido la que ha cambiado. Cada una de las teclas tiene asociado un numero diferente para que el controlador de teclado pueda reconocerlas (nos referimos al scancode) y depende únicamente de la posición que la tecla ocupa en le teclado. En resumen, cuando una tecla es presionada o liberada, el controlador de teclado envía scancodes, códigos exploración del teclado, al driver del teclado (keyboard.c).

Para permitir que varias teclas sean pulsadas simultáneamente, el teclado genera un código diferente cuando una tecla se pulsa y cuando dicha tecla se libera.

El núcleo cuando recibe los scancodes realiza una traducción a un código interno denominado keycodes (código resultante deseado) que luego se le pasa al keymap para que este devuelva el carácter o la secuencia encontrada o la acción allí descrita al programa de usuario.

Los keymaps son mapas de caracteres empleados para determinar el código de carácter que se le pasa a la aplicación basándose en la tecla que ha sido pulsada y los modificadores activos en ese momento.

Puede haber varios keymaps y la razón de esto es que no todos los lenguajes poseen los mismos caracteres y símbolos de puntuación.

ScanCodes

Existen dos tipos; Make Codes (pulsación) y Break Codes (liberación). El tamaño es de 8 bits, el MSB identifica pulsación y liberación. Como consecuencia existen 128 teclas distintas como máximo. El Teclado detecta tanto pulsaciones como liberaciones.

Keymaps Los keymap son mapas de caracteres empleados para determinar el código de carácter que se le pasa a la aplicación basándose en la tecla que ha sido pulsada y los modificadores activos en ese momento.

Scancodes & Keycodes Al presionar o soltar una tecla se producen secuencias de 1 a 6 bytes conocidas como scancodes, que el núcleo tiene que asociar a keycodes.

Para conseguir esto cada tecla se asocia a un keycode único k en un rango de 1 hasta 127, y el presionar la tecla k produce el keycode k, mientras que al soltarla produce el keycode k+128.

Teclado 3. Estructuras de Datos

struct tty_struct: Estructura que se utiliza como manejador genérico del terminal. Se encuentra en include/linux/tty.

```
182/*
183 * Where all of the state associated with a tty is kept while the tty
184 * is open.  Since the termios state should be kept even if the tty
185 * has been closed --- for things like the baud rate, etc --- it is
186 * not stored here, but rather a pointer to the real state is stored
187 * here.  Possible the winsize structure should have the same
```

```
188 * treatment, but (1) the default 80x24 is usually right and (2) it's
189 * most often used by a windowing system, which will set the correct
190 * size each time the window is created or resized anyway.
191 *                                      - TYT, 9/14/92
192 */
193struct tty_struct {
194        int     magic;
195        struct tty_driver *driver;
196        int index;
197        struct tty_ldisc ldisc;
198        struct mutex termios_mutex;
199        struct ktermios *termios, *termios_locked;
200        char name[64];
201        struct pid *pgrp;
202        struct pid *session;
203        unsigned long flags;
204        int count;
205        struct winsize winsize;
206        unsigned char stopped:1, hw_stopped:1, flow_stopped:1, packet:1;
207        unsigned char low_latency:1, warned:1;
208        unsigned char ctrl_status;
209        unsigned int receive_room;      /* Bytes free for queue */
210
211        struct tty_struct *link;
212        struct fasync_struct *fasync;
213        struct tty_bufhead buf;
214        int alt_speed;          /* For magic substitution of 38400 bps */
215        wait_queue_head_t write_wait;
216        wait_queue_head_t read_wait;
217        struct work_struct hangup_work;
218        void *disc_data;
219        void *driver_data;
220        struct list_head tty_files;
221
222#define N_TTY_BUF_SIZE 4096
223
224        /*
225         * The following is data for the N_TTY line discipline.  For
226         * historical reasons, this is included in the tty structure.
227         */
228        unsigned int column;
229        unsigned char lnext:1, erasing:1, raw:1, real_raw:1, icanon:1;
230        unsigned char closing:1;
231        unsigned short minimum_to_wake;
232        unsigned long overrun_time;
233        int num_overrun;
234        unsigned long process_char_map[256/(8*sizeof(unsigned long))];
235        char *read_buf;
236        int read_head;
237        int read_tail;
238        int read_cnt;
239        unsigned long read_flags[N_TTY_BUF_SIZE/(8*sizeof(unsigned long))];
240        int canon_data;
241        unsigned long canon_head;
242        unsigned int canon_column;
243        struct mutex atomic_read_lock;
244        struct mutex atomic_write_lock;
245        unsigned char *write_buf;
```

```
246     int write_cnt;
247     spinlock_t read_lock;
248     /* If the tty has a pending do_SAK, queue it here - akpm */
249     struct work_struct SAK_work;
250};
```

struct input_dev: Estructura que almacena el valor devuelto por el manejador de ese dispositivo. Se encuentra en include/linux/input.h.

```
struct input_dev {
937
938     void *private;
939
940     const char *name;
941     const char *phys;
942     const char *uniq;
943     struct input_id id;
944
945     unsigned long evbit[NBITS(EV_MAX)];
946     unsigned long keybit[NBITS(KEY_MAX)];
947     unsigned long relbit[NBITS(REL_MAX)];
948     unsigned long absbit[NBITS(ABS_MAX)];
949     unsigned long mscbit[NBITS(MSC_MAX)];
950     unsigned long ledbit[NBITS(LED_MAX)];
951     unsigned long sndbit[NBITS(SND_MAX)];
952     unsigned long ffbit[NBITS(FF_MAX)];
953     unsigned long swbit[NBITS(SW_MAX)];
954
955     unsigned int keycodemax;
956     unsigned int keycodesize;
957     void *keycode;
958     int (*setkeycode)(struct input_dev *dev, int scancode, int keycode);
959     int (*getkeycode)(struct input_dev *dev, int scancode, int *keycode);
960
961     struct ff_device *ff;
962
963     unsigned int repeat_key;
964     struct timer_list timer;
965
966     int state;
967
968     int sync;
969
970     int abs[ABS_MAX + 1];
971     int rep[REP_MAX + 1];
972
973     unsigned long key[NBITS(KEY_MAX)];
974     unsigned long led[NBITS(LED_MAX)];
975     unsigned long snd[NBITS(SND_MAX)];
976     unsigned long sw[NBITS(SW_MAX)];
977
```

```
 978        int absmax[ABS_MAX + 1];
 979        int absmin[ABS_MAX + 1];
 980        int absfuzz[ABS_MAX + 1];
 981        int absflat[ABS_MAX + 1];
 982
 983        int (*open)(struct input_dev *dev);
 984        void (*close)(struct input_dev *dev);
 985        int (*flush)(struct input_dev *dev, struct file *file);
 986        int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);
 987
 988        struct input_handle *grab;
 989
 990        struct mutex mutex;     /* serializes open and close operations */
 991        unsigned int users;
 992
 993        struct device dev;
 994        union {                 /* temporarily so while we switching to struct device */
 995                struct device *dev;
 996        } cdev;
 997
 998        struct list_head        h_list;
 999        struct list_head        node;
1000};
```

struct input_handle: Estructura del manejador de dispositivo. Se encuentra en include/linux/input.h.

```
1052struct input_handle;
1053
1054/**
1055 * struct input_handler - implements one of interfaces for input devices
1056 * @private: driver-specific data
1057 * @event: event handler
1058 * @connect: called when attaching a handler to an input device
1059 * @disconnect: disconnects a handler from input device
1060 * @start: starts handler for given handle. This function is called by
1061 *      input core right after connect() method and also when a process
1062 *      that "grabbed" a device releases it
1063 * @fops: file operations this driver implements
1064 * @minor: beginning of range of 32 minors for devices this driver
1065 *      can provide
1066 * @name: name of the handler, to be shown in /proc/bus/input/handlers
1067 * @id_table: pointer to a table of input_device_ids this driver can
1068 *      handle
1069 * @blacklist: prointer to a table of input_device_ids this driver should
1070 *      ignore even if they match @id_table
1071 * @h_list: list of input handles associated with the handler
1072 * @node: for placing the driver onto input_handler_list
1073 */
1074struct input_handler {
1075
1076        void *private;
1077
1078        void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
```

```
1079        int (*connect)(struct input_handler *handler, struct input_dev *dev, const struct
input_device_id *id);
1080        void (*disconnect)(struct input_handle *handle);
1081        void (*start)(struct input_handle *handle);
1082
1083        const struct file_operations *fops;
1084        int minor;
1085        const char *name;
1086
1087        const struct input_device_id *id_table;
1088        const struct input_device_id *blacklist;
1089
1090        struct list_head        h_list;
1091        struct list_head        node;
1092};
1093
1094struct input_handle {
1095
1096        void *private;
1097
1098        int open;
1099        const char *name;
1100
1101        struct input_dev *dev;
1102        struct input_handler *handler;
1103
1104        struct list_head        d_node;
1105        struct list_head        h_node;
1106};
```

struct vc_data: Estructura de datos referente a la consola. Se encuentra en include/linux/console_struct.h

```
 23struct vc_data {
 24        unsigned short  vc_num;                 /* Console number */
 25        unsigned int    vc_cols;                /* [#] Console size */
 26        unsigned int    vc_rows;
 27        unsigned int    vc_size_row;            /* Bytes per row */
 28        unsigned int    vc_scan_lines;          /* # of scan lines */
 29        unsigned long   vc_origin;              /* [!] Start of real screen */
 30        unsigned long   vc_scr_end;             /* [!] End of real screen */
 31        unsigned long   vc_visible_origin;      /* [!] Top of visible window */
 32        unsigned int    vc_top, vc_bottom;      /* Scrolling region */
 33        const struct consw *vc_sw;
 34        unsigned short  *vc_screenbuf;          /* In-memory character/attribute buffer */
 35        unsigned int    vc_screenbuf_size;
 36        unsigned char   vc_mode;                /* KD_TEXT, ... */
 37        /* attributes for all characters on screen */
 38        unsigned char   vc_attr;                /* Current attributes */
 39        unsigned char   vc_def_color;           /* Default colors */
 40        unsigned char   vc_color;               /* Foreground & background */
 41        unsigned char   vc_s_color;             /* Saved foreground & background */
 42        unsigned char   vc_ulcolor;             /* Color for underline mode */
 43        unsigned char   vc_itcolor;
 44        unsigned char   vc_halfcolor;           /* Color for half intensity mode */
```

```
 45        /* cursor */
 46        unsigned int    vc_cursor_type;
 47        unsigned short  vc_complement_mask;     /* [#] Xor mask for mouse pointer */
 48        unsigned short  vc_s_complement_mask;   /* Saved mouse pointer mask */
 49        unsigned int    vc_x, vc_y;             /* Cursor position */
 50        unsigned int    vc_saved_x, vc_saved_y;
 51        unsigned long   vc_pos;                 /* Cursor address */
 52        /* fonts */
 53        unsigned short  vc_hi_font_mask;        /* [#] Attribute set for upper 256 chars of font or 0 if
not supported */
 54        struct console_font vc_font;            /* Current VC font set */
 55        unsigned short  vc_video_erase_char;    /* Background erase character */
 56        /* VT terminal data */
 57        unsigned int    vc_state;               /* Escape sequence parser state */
 58        unsigned int    vc_npar,vc_par[NPAR];   /* Parameters of current escape sequence */
 59        struct tty_struct *vc_tty;              /* TTY we are attached to */
 60        /* data for manual vt switching */
 61        struct vt_mode  vt_mode;
 62        struct pid      *vt_pid;
 63        int             vt_newvt;
 64        wait_queue_head_t paste_wait;
 65        /* mode flags */
 66        unsigned int    vc_charset      : 1;    /* Character set G0 / G1 */
 67        unsigned int    vc_s_charset    : 1;    /* Saved character set */
 68        unsigned int    vc_disp_ctrl    : 1;    /* Display chars < 32? */
 69        unsigned int    vc_toggle_meta  : 1;    /* Toggle high bit? */
 70        unsigned int    vc_decscnm      : 1;    /* Screen Mode */
 71        unsigned int    vc_decom        : 1;    /* Origin Mode */
 72        unsigned int    vc_decawm       : 1;    /* Autowrap Mode */
 73        unsigned int    vc_deccm        : 1;    /* Cursor Visible */
 74        unsigned int    vc_decim        : 1;    /* Insert Mode */
 75        unsigned int    vc_deccolm      : 1;    /* 80/132 Column Mode */
 76        /* attribute flags */
 77        unsigned int    vc_intensity    : 2;    /* 0=half-bright, 1=normal, 2=bold */
 78        unsigned int    vc_italic:1;
 79        unsigned int    vc_underline    : 1;
 80        unsigned int    vc_blink        : 1;
 81        unsigned int    vc_reverse      : 1;
 82        unsigned int    vc_s_intensity  : 2;    /* saved rendition */
 83        unsigned int    vc_s_italic:1;
 84        unsigned int    vc_s_underline  : 1;
 85        unsigned int    vc_s_blink      : 1;
 86        unsigned int    vc_s_reverse    : 1;
 87        /* misc */
 88        unsigned int    vc_ques         : 1;
 89        unsigned int    vc_need_wrap    : 1;
 90        unsigned int    vc_can_do_color : 1;
 91        unsigned int    vc_report_mouse : 2;
 92        unsigned int    vc_kmalloced    : 1;
 93        unsigned char   vc_utf          : 1;    /* Unicode UTF-8 encoding */
 94        unsigned char   vc_utf_count;
 95                int     vc_utf_char;
 96        unsigned int    vc_tab_stop[8];         /* Tab stops. 256 columns. */
 97        unsigned char   vc_palette[16*3];       /* Colour palette for VGA+ */
 98        unsigned short * vc_translate;
 99        unsigned char   vc_G0_charset;
100         unsigned char   vc_G1_charset;
101         unsigned char   vc_saved_G0;
```

```
102        unsigned char   vc_saved_G1;
103        unsigned int    vc_bell_pitch;          /* Console bell pitch */
104        unsigned int    vc_bell_duration;       /* Console bell duration */
105        struct vc_data **vc_display_fg;         /* [!] Ptr to var holding fg console for this display */
106        unsigned long   vc_uni_pagedir;
107        unsigned long   *vc_uni_pagedir_loc;  /* [!] Location of uni_pagedir variable for this console
*/
108        /* additional information is in vt_kern.h */
109};
```

struct pt_regs: Estructura que hace referencia a los registros de la CPU que se encuentra en el archivo ptrace.h

```
29struct pt_regs {
30        unsigned long gpr[32];
31        unsigned long nip;
32        unsigned long msr;
33        unsigned long orig_gpr3;        /* Used for restarting system calls */
34        unsigned long ctr;
35        unsigned long link;
36        unsigned long xer;
37        unsigned long ccr;
38#ifdef __powerpc64__
39        unsigned long softe;            /* Soft enabled/disabled */
40#else
41        unsigned long mq;               /* 601 only (not used at present) */
42                                        /* Used on APUS to hold IPL value. */
43#endif
44        unsigned long trap;             /* Reason for being here */
45        /* N.B. for critical exceptions on 4xx, the dar and dsisr
46           fields are overloaded to hold srr0 and srr1. */
47        unsigned long dar;              /* Fault registers */
48        unsigned long dsisr;            /* on 4xx/Book-E used for ESR */
49        unsigned long result;           /* Result of a system call */
50};
```

Teclado 4. Funciones

getkeycode

Dado un scancode, devuelve su keycode correspondiente.

```
170int getkeycode(unsigned int scancode)
171{
172        struct input_handle *handle;
173        int keycode;
174        int error = -ENODEV;
175
176        list_for_each_entry(handle, &kbd_handler.h_list, h_node) {
177                error = handle->dev->getkeycode(handle->dev, scancode, &keycode);
178                if (!error)
179                        return keycode;
180        }
181
182        return error;
183}
```

setkeycode

Permite cambiar la asociación entre scancodes y keycodes. Con ello podemos crear diferentes mapas de caracteres.

```
185int setkeycode(unsigned int scancode, unsigned int keycode)
186{
187        struct input_handle *handle;
188        int error = -ENODEV;
189
190        list_for_each_entry(handle, &kbd_handler.h_list, h_node) {
191                error = handle->dev->setkeycode(handle->dev, scancode, keycode);
192                if (!error)
193                        break;
194        }
195
196        return error;
197}
```

put_queue

Inserta un carácter en el buffer del terminal.

```
300 static void put_queue(struct vc_data *vc, int ch)
301 {
302         struct tty_struct *tty = vc->vc_tty;
303
304         if (tty) {
305                 tty_insert_flip_char(tty, ch, 0);
306                 con_schedule_flip(tty);
307         }
308 }
```

puts_queue

Inserta una ristra en el buffer del terminal.

```
310 static void puts_queue(struct vc_data *vc, char *cp)
311 {
312         struct tty_struct *tty = vc->vc_tty;
313
314         if (!tty)
315                 return;
316
317         while (*cp) {
318                 tty_insert_flip_char(tty, *cp, 0);
319                 cp++;
320         }
321         con_schedule_flip(tty);
322 }
```

applkey

Inserta el código de la tecla en el buffer del terminal.

```
324 static void applkey(struct vc_data *vc, int key, char mode)
325 {
326         static char buf[] = { 0x1b, 'O', 0x00, 0x00 };
327
328         buf[1] = (mode ? 'O' : '[');
329         buf[2] = key;
330         puts_queue(vc, buf);
331 }
```

to_utf8

Inserta un carácter en el terminal tras pasarlo a Unicode (con un tamaño de 16 bits).

```
314static void to_utf8(struct vc_data *vc, uint c)
315{
316       if (c < 0x80)
317              /*  0******* */
318              put_queue(vc, c);
319       else if (c < 0x800) {
320              /* 110***** 10****** */
321              put_queue(vc, 0xc0 | (c >> 6));
322              put_queue(vc, 0x80 | (c & 0x3f));
323       } else if (c < 0x10000) {
324              if (c >= 0xD800 && c < 0xE000)
325                     return;
326              if (c == 0xFFFF)
327                     return;
328              /* 1110**** 10****** 10****** */
329              put_queue(vc, 0xe0 | (c >> 12));
330              put_queue(vc, 0x80 | ((c >> 6) & 0x3f));
331              put_queue(vc, 0x80 | (c & 0x3f));
332       } else if (c < 0x110000) {
333              /* 11110*** 10****** 10****** 10****** */
334              put_queue(vc, 0xf0 | (c >> 18));
335              put_queue(vc, 0x80 | ((c >> 12) & 0x3f));
336              put_queue(vc, 0x80 | ((c >> 6) & 0x3f));
337              put_queue(vc, 0x80 | (c & 0x3f));
338       }
339}
```

handle_diacr

Se utiliza para combinar las teclas especiales (“¨”, “´”) con una tecla ordinaria para obtener caracteres especiales. Si se combina con el espacio se muestra la tecla especial pulsada.

```
386static unsigned int handle_diacr(struct vc_data *vc, unsigned int ch)
387{
388       unsigned int d = diacr;
389       unsigned int i;
390
391       diacr = 0;
392
393       if ((d & ~0xff) == BRL_UC_ROW) {
394              if ((ch & ~0xff) == BRL_UC_ROW)
395                     return d | ch;
396       } else {
397              for (i = 0; i < accent_table_size; i++)
398                     if (accent_table[i].diacr == d && accent_table[i].base == ch)
399                            return accent_table[i].result;
400       }
401
402       if (ch == ' ' || ch == (BRL_UC_ROW|0) || ch == d)
403              return d;
404
405       if (kbd->kbdmode == VC_UNICODE)
```

```
406             to_utf8(vc, conv_8bit_to_uni(d));
407         else if (d < 0x100)
408             put_queue(vc, d);
409
410         return ch;
411}
```

fn_enter

Introduce en el buffer del terminal el carácter de retorno de carro (cuando pulsamos la tecla ENTER). También le pude añadir el LF: nueva línea.

```
416static void fn_enter(struct vc_data *vc)
417{
418         if (diacr) {
419             if (kbd->kbdmode == VC_UNICODE)
420                 to_utf8(vc, conv_8bit_to_uni(diacr));
421             else if (diacr < 0x100)
422                 put_queue(vc, diacr);
423             diacr = 0;
424         }
425         put_queue(vc, 13);
426         if (vc_kbd_mode(kbd, VC_CRLF))
427             put_queue(vc, 10);
428}
```

fn_caps_toggle

Intercambia el estado del led de Bloq. Mayús.

```
446 static void fn_caps_toggle(struct vc_data *vc)
447 {
448         if (rep)
449             return;
450         chg_vc_kbd_led(kbd, VC_CAPSLOCK);
451 }

131 static inline void chg_vc_kbd_led(struct kbd_struct * kbd, int flag)
132 {
133         kbd->ledflagstate ^= 1 << flag;
134 }
```

fn_caps_on

Activa el led de Bloq. Mayús.

```
453 static void fn_caps_on(struct vc_data *vc)
454 {
455         if (rep)
456             return;
457         set_vc_kbd_led(kbd, VC_CAPSLOCK);
458 }
```

```
101 static inline void set_vc_kbd_led(struct kbd_struct * kbd, int flag)
102 {
103         kbd->ledflagstate |= 1 << flag;
104 }
```

fn_show_ptregs

Muestra el contenido de los registros del procesador.

```
460 static void fn_show_ptregs(struct vc_data *vc)

461 {
462         struct pt_regs *regs = get_irq_regs();
463         if (regs)
464                 show_regs(regs);
465 }
```

fn_hold

Inicia o detiene la consola. Cuando la detiene desactiva el Bloq. Despl.

```
467 static void fn_hold(struct vc_data *vc)
468 {
469         struct tty_struct *tty = vc->vc_tty;
470
471         if (rep || !tty)
472                 return;
473                   /*...*/
479         if (tty->stopped)
480                 start_tty(tty);
481         else
482                 stop_tty(tty);
483 }
```

fn_num / fn_bare_num

Para cambiar el led del teclado numérico.

```
485 static void fn_num(struct vc_data *vc)
486 {
487         if (vc_kbd_mode(kbd,VC_APPLIC))
488                 applkey(vc, 'P', 1);
489         else
490                 fn_bare_num(vc);
491 }

499 static void fn_bare_num(struct vc_data *vc)
500 {
501         if (!rep)
502                 chg_vc_kbd_led(kbd, VC_NUMLOCK);
503 }
```

fn_lastcons

Para cambiar a la última consola utilizada.

```
505 static void fn_lastcons(struct vc_data *vc)
506 {
507         /* switch to the last used console, ChN */
508         set_console(last_console);
509 }
```

fn_dec_console

Cambia a la consola que precede a la actual.

```
511 static void fn_dec_console(struct vc_data *vc)
512 {
513         int i, cur = fg_console;
514
515         /* Currently switching?  Queue this next switch relative to that. */
516         if (want_console != -1)
517                 cur = want_console;
518
519         for (i = cur - 1; i != cur; i--) {
520                 if (i == -1)
521                         i = MAX_NR_CONSOLES - 1;
522                 if (vc_cons_allocated(i))
523                         break;
524         }
525         set_console(i);
526 }
```

fn_inc_console

Función simétrica a la anterior, en este caso se pasa a la consola que sigue a la actual.

```
528 static void fn_inc_console(struct vc_data *vc)
529 {
530         int i, cur = fg_console;
531
532         /* Currently switching?  Queue this next switch relative to that. */
533         if (want_console != -1)
534                 cur = want_console;
535
536         for (i = cur+1; i != cur; i++) {
537                 if (i == MAX_NR_CONSOLES)
538                         i = 0;
539                 if (vc_cons_allocated(i))
540                         break;
541         }
542         set_console(i);
543 }
```

K_cons

Se usa para especificar a qué consola cambiar. Por ejemplo cuando usamos Control+Alt+F<n>.

```
690 static void k_cons(struct vc_data *vc, unsigned char value, char up_flag)
691 {
692         if (up_flag)
693                 return;
694         set_console(value);
695 }
```

fn_scroll_forw

Scroll hacia abajo de la consola (la mitad).

```
555 static void fn_scroll_forw(struct vc_data *vc)
556 {
557         scrollfront(vc, 0);
558 }

987 void scrollfront(struct vc_data *vc, int lines)
988 {
989         if (!lines)
990                 lines = vc->vc_rows / 2;
991         scrolldelta(lines);
992 }
```

fn_scroll_back

Scroll hacia arriba de la consola (la mitad).

```
560 static void fn_scroll_back(struct vc_data *vc)
561 {
562         scrollback(vc, 0);
563 }

980 void scrollback(struct vc_data *vc, int lines)
981 {
982         if (!lines)
983                 lines = vc->vc_rows / 2;
984         scrolldelta(-lines);
985 }
```

fn_boot_it

Se usa para enviar a la consola la combinación de teclas Control+Alt+Supr. Si la variable C_A_D está activada se reinicia, si no, se envía una señal al proceso activo para que se haga cargo de la situación.

```
575 static void fn_boot_it(struct vc_data *vc)
576 {
577         ctrl_alt_del();
578 }

900 void ctrl_alt_del(void)
901 {
902         static DECLARE_WORK(cad_work, deferred_cad);
903
904         if (C_A_D)
905                 schedule_work(&cad_work);
906         else
907                 kill_cad_pid(SIGINT, 1);
908 }
```

fn_compose

Se utiliza para detectar combinaciones de teclas que producirán caracteres especiales, como puede ser la cedilla. (Control, p.e.)

```
580 static void fn_compose(struct vc_data *vc)
581 {
582         dead_key_next = 1;
583 }
```

fn_spawn_con

Con esta función podemos crear una nueva consola, usando el PID del proceso actual. La operación se realiza en exclusión mutua.

```
585 static void fn_spawn_con(struct vc_data *vc)
586 {
587         spin_lock(&vt_spawn_con.lock);
588         if (vt_spawn_con.pid)
589                 if (kill_pid(vt_spawn_con.pid, vt_spawn_con.sig, 1)) {
590                         put_pid(vt_spawn_con.pid);
591                         vt_spawn_con.pid = NULL;
592                 }
593         spin_unlock(&vt_spawn_con.lock);
594 }
```

fn_SAK

Esta función es usada en caso de “desastre”, llamando a algunas funciones de recuperación en caso de que la consola no responda como puede ser “reset_vc”

que reseteará la consola y la reiniciará a su estado por defecto. Esta función está definida en Linux/drivers/char/vt_ioctl.c

```
580static void fn_SAK(struct vc_data *vc)
581{
582	struct work_struct *SAK_work = &vc_cons[fg_console].SAK_work;
583	schedule_work(SAK_work);
584}
```

K_spec

Facilita la llamada de las funciones del manejador, agrupándolas en un vector, y permitiendo su llamada pasándole un índice.

```
621 static void k_spec(struct vc_data *vc, unsigned char value, char up_flag)
622 {
623	if (up_flag)
624		return;
625	if (value >= ARRAY_SIZE(fn_handler))
626		return;
627	if ((kbd->kbdmode == VC_RAW ||
628	     kbd->kbdmode == VC_MEDIUMRAW) &&
629	     value != KVAL(K_SAK))
630		return;	/* SAK is allowed even in raw mode */
631	fn_handler[value](vc);
632 }
```

k_unicode

Inserta las teclas muertas en el buffer del terminal, pasándolas a UTF-8 antes si fuera necesario.

```
639 static void k_unicode(struct vc_data *vc, unsigned int value, char up_flag)
640 {
641	if (up_flag)
642		return;	/* no action, if this is a key release */
643
644	if (diacr)
645		value = handle_diacr(vc, value);
646
647	if (dead_key_next) {
648		dead_key_next = 0;
649		diacr = value;
650		return;
651	}
652	if (kbd->kbdmode == VC_UNICODE)
653		to_utf8(vc, conv_8bit_to_uni(value));
654	else if (value < 0x100)
655		put_queue(vc, value);
656 }
```

K_dead2

Permite combinar las teclas muertas para modificar la siguiente tecla.

```
675 static void k_dead2(struct vc_data *vc, unsigned char value, char up_flag)
676 {
677         k_deadunicode(vc, value, up_flag);
678 }

663 static void k_deadunicode(struct vc_data *vc, unsigned int value, char up_flag)
664 {
665         if (up_flag)
666                 return;
667         diacr = (diacr ? handle_diacr(vc, value) : value);
668 }
```

k_fn

Nos permitirá usar las teclas de función (<F1>,<F2>...).

```
697 static void k_fn(struct vc_data *vc, unsigned char value, char up_flag)
698 {
699         unsigned v;
700
701         if (up_flag)
702                 return;
703         v = value;
704         if (v < ARRAY_SIZE(func_table)) {
705                 if (func_table[value])
706                         puts_queue(vc, func_table[value]);
707         } else
708                 printk(KERN_ERR "k_fn called with value=%d\n", value);
709 }
```

k_cur

Gracias a esta función podremos usar los cursores.

```
711 static void k_cur(struct vc_data *vc, unsigned char value, char up_flag)
712 {
713         static const char cur_chars[] = "BDCA";
714
715         if (up_flag)
716                 return;
717         applkey(vc, cur_chars[value], vc_kbd_mode(kbd, VC_CKMODE));
718 }
```

K_pad

Si el Bloq. Num. está activado se insertarán en el buffer los números, si no, se ejecutarán las diferentes funciones asociadas a las distintas teclas (Inicio, Fin, Ins, etc.).

```
static void k_pad(struct vc_data *vc, unsigned char value, char up_flag)
 698{
 699        static const char pad_chars[] = "0123456789+-*/\015,.?()#";
 700        static const char app_map[] = "pqrstuvwxylSRQMnnmPQS";
 701
 702        if (up_flag)
 703                return;         /* no action, if this is a key release */
 704
 705        /* kludge... shift forces cursor/number keys */
 706        if (vc_kbd_mode(kbd, VC_APPLIC) && !shift_down[KG_SHIFT]) {
 707                applkey(vc, app_map[value], 1);
 708                return;
 709        }
 710
 711        if (!vc_kbd_led(kbd, VC_NUMLOCK))
 712                switch (value) {
 713                        case KVAL(K_PCOMMA):
 714                        case KVAL(K_PDOT):
 715                                k_fn(vc, KVAL(K_REMOVE), 0);
 716                                return;
 717                        case KVAL(K_P0):
 718                                k_fn(vc, KVAL(K_INSERT), 0);
 719                                return;
 720                        case KVAL(K_P1):
 721                                k_fn(vc, KVAL(K_SELECT), 0);
 722                                return;
 723                        case KVAL(K_P2):
 724                                k_cur(vc, KVAL(K_DOWN), 0);
 725                                return;
 726                        case KVAL(K_P3):
 727                                k_fn(vc, KVAL(K_PGDN), 0);
 728                                return;
 729                        case KVAL(K_P4):
 730                                k_cur(vc, KVAL(K_LEFT), 0);
 731                                return;
 732                        case KVAL(K_P6):
 733                                k_cur(vc, KVAL(K_RIGHT), 0);
 734                                return;
 735                        case KVAL(K_P7):
 736                                k_fn(vc, KVAL(K_FIND), 0);
 737                                return;
 738                        case KVAL(K_P8):
 739                                k_cur(vc, KVAL(K_UP), 0);
 740                                return;
 741                        case KVAL(K_P9):
 742                                k_fn(vc, KVAL(K_PGUP), 0);
 743                                return;
 744                        case KVAL(K_P5):
 745                                applkey(vc, 'G', vc_kbd_mode(kbd, VC_APPLIC));
 746                                return;
 747                }
 748
 749        put_queue(vc, pad_chars[value]);
```

```
750       if (value == KVAL(K_PENTER) && vc_kbd_mode(kbd, VC_CRLF))
751               put_queue(vc, 10);
```

K_shift

Inserta en el buffer la combinación Shift+Tecla, controlando que las dos teclas Shift se pulsen o liberen al mismo tiempo.

```
static void k_shift(struct vc_data *vc, unsigned char value, char up_flag)
 755{
 756       int old_state = shift_state;
 757
 758       if (rep)
 759               return;
 760       /*
 761        * Mimic typewriter:
 762        * a CapsShift key acts like Shift but undoes CapsLock
 763        */
 764       if (value == KVAL(K_CAPSSHIFT)) {
 765               value = KVAL(K_SHIFT);
 766               if (!up_flag)
 767                       clr_vc_kbd_led(kbd, VC_CAPSLOCK);
 768       }
 769
 770       if (up_flag) {
 771               /*
 772                * handle the case that two shift or control
 773                * keys are depressed simultaneously
 774                */
 775               if (shift_down[value])
 776                       shift_down[value]--;
 777       } else
 778               shift_down[value]++;
 779
 780       if (shift_down[value])
 781               shift_state |= (1 << value);
 782       else
 783               shift_state &= ~(1 << value);
 784
 785       /* kludge */
 786       if (up_flag && shift_state != old_state && npadch != -1) {
 787               if (kbd->kbdmode == VC_UNICODE)
 788                       to_utf8(vc, npadch);
 789               else
 790                       put_queue(vc, npadch & 0xff);
 791               npadch = -1;
 792       }
 793}
```

k_meta

Inserta en el buffer la combinación Alt+Tecla.

```
818 static void k_meta(struct vc_data *vc, unsigned char value, char up_flag)
819 {
820         if (up_flag)
821                 return;
822
823         if (vc_kbd_mode(kbd, VC_META)) {
824                 put_queue(vc, '\033');
825                 put_queue(vc, value);
826         } else
827                 put_queue(vc, value | 0x80);
828 }
```

K_ascii

Con esta función podremos combinar la tecla “Alt” con un código numérico para producir el correspondiente carácter.

```
830 static void k_ascii(struct vc_data *vc, unsigned char value, char up_flag)
831 {
832         int base;
833
834         if (up_flag)
835                 return;
836
837         if (value < 10) {
838                 /* decimal input of code, while Alt depressed */
839                 base = 10;
840         } else {
841                 /* hexadecimal input of code, while AltGr depressed */
842                 value -= 10;
843                 base = 16;
844         }
845
846         if (npadch == -1)
847                 npadch = value;
848         else
849                 npadch = npadch * base + value;
850 }
```

K_lock

Establece el estado del correspondiente modificador de bloqueo de las teclas (Control, Alt, AltGr, Shift). Tanto derecha como izquierda.

```
852 static void k_lock(struct vc_data *vc, unsigned char value, char up_flag)
853 {
854         if (up_flag || rep)
855                 return;
856         chg_vc_kbd_lock(kbd, value);
857 }

116 static inline void chg_vc_kbd_lock(struct kbd_struct * kbd, int flag)
```

```
117 {
118         kbd->lockstate ^= 1 << flag;
119 }
```

getledstate

Devuelve el estado de los leds del teclado.

```
951 unsigned char getledstate(void)
952 {
953         return ledstate;
954 }
```

setledstate

Con esta función podemos especificar el estado de los leds del teclado.

```
956 void setledstate(struct kbd_struct *kbd, unsigned int led)
957 {
958         if (!(led & ~7)) {
959                 ledioctl = led;
960                 kbd->ledmode = LED_SHOW_IOCTL;
961         } else
962                 kbd->ledmode = LED_SHOW_FLAGS;
963         set_leds();
964 }
```

getleds

Esta función nos devuelve los leds presentes en el teclado.

```
966 static inline unsigned char getleds(void)
967 {
968         struct kbd_struct *kbd = kbd_table + fg_console;
969         unsigned char leds;
970         int i;
971
972         if (kbd->ledmode == LED_SHOW_IOCTL)
973                 return ledioctl;
974
975         leds = kbd->ledflagstate;
976
977         if (kbd->ledmode == LED_SHOW_MEM) {
978                 for (i = 0; i < 3; i++)
979                         if (ledptrs[i].valid) {
980                                 if (*ledptrs[i].addr & ledptrs[i].mask)
981                                         leds |= (1 << i);
982                                 else
983                                         leds &= ~(1 << i);
984                         }
985         }
986         return leds;
987 }
```

kbd_handler

Estructura del Manejador de teclado con sus operaciones asociadas.

```
1362 static struct input_handler kbd_handler = {
1363         .event          = kbd_event,
1364         .connect        = kbd_connect,
1365         .disconnect     = kbd_disconnect,
1366         .start          = kbd_start,
1367         .name           = "kbd",
1368         .id_table       = kbd_ids,
1369 };
```

kbd_connect

Se llama cuando un teclado se conecta por primera vez al ordenador. Rellena la estructura del manejador para poder recibir eventos de entrada a continuación.

```
1268static int kbd_connect(struct input_handler *handler, struct input_dev *dev,
1269                        const struct input_device_id *id)
1270{
1271         struct input_handle *handle;
1272         int error;
1273         int i;
1274
1275         for (i = KEY_RESERVED; i < BTN_MISC; i++)
1276                 if (test_bit(i, dev->keybit))
1277                         break;
1278
1279         if (i == BTN_MISC && !test_bit(EV_SND, dev->evbit))
1280                 return -ENODEV;
1281
1282         handle = kzalloc(sizeof(struct input_handle), GFP_KERNEL);
1283         if (!handle)
1284                 return -ENOMEM;
1285
1286         handle->dev = dev;
1287         handle->handler = handler;
1288         handle->name = "kbd";
1289
1290         error = input_register_handle(handle);
1291         if (error)
1292                 goto err_free_handle;
1293
1294         error = input_open_device(handle);
1295         if (error)
1296                 goto err_unregister_handle;
1297
1298         return 0;
1299
1300 err_unregister_handle:
1301         input_unregister_handle(handle);
1302 err_free_handle:
```

```
1303        kfree(handle);
1304        return error;
1305}
```

kbd_disconnect

Se llama cuando se desconecta el teclado. Libera la memoria para el manejador asociado.

```
1307static void kbd_disconnect(struct input_handle *handle)
1308{
1309        input_close_device(handle);
1310        input_unregister_handle(handle);
1311        kfree(handle);
1312}
```

kbd_start

Inicia el controlador de teclado refrescando el estado de los LEDs para igualar el estado del sistema.

```
1318static void kbd_start(struct input_handle *handle)
1319{
1320        unsigned char leds = ledstate;
1321
1322        tasklet_disable(&keyboard_tasklet);
1323        if (leds != 0xff) {
1324                input_inject_event(handle, EV_LED, LED_SCROLLL, !!(leds & 0x01));
1325                input_inject_event(handle, EV_LED, LED_NUML,    !!(leds & 0x02));
1326                input_inject_event(handle, EV_LED, LED_CAPSL,   !!(leds & 0x04));
1327                input_inject_event(handle, EV_SYN, SYN_REPORT, 0);
1328        }
1329        tasklet_enable(&keyboard_tasklet);
1330}
```

kbd_event

Cada vez que se pulsa una tecla, o una combinación de ellas, se llama a kbd_event; que en función de la configuración del teclado, inserta el carácter de una forma u otra.

```
1250 static void kbd_event(struct input_handle *handle, unsigned int event_type,
1251                        unsigned int event_code, int value)
1252{
1253        if (event_type == EV_MSC && event_code == MSC_RAW && HW_RAW(handle->dev))
1254                kbd_rawcode(value);
1255        if (event_type == EV_KEY)
1256                kbd_keycode(event_code, value, HW_RAW(handle->dev));
1257        tasklet_schedule(&keyboard_tasklet);
```

```
1258        do_poke_blanked_console = 1;
1259        schedule_console_callback();
1260}
```

kbd_rawcode

Inserta los caracteres en el buffer en modo RAW, es decir, sin traducir los scancodes a keycodes.

```
1130 static void kbd_rawcode(unsigned char data)
1131 {
1132         struct vc_data *vc = vc_cons[fg_console].d;
1133         kbd = kbd_table + fg_console;
1134         if (kbd->kbdmode == VC_RAW)
1135                 put_queue(vc, data);
1136 }
```

kbd_keycode

Inserta el keycode en el buffer, eligiendo la función correspondiente, alguna de las vistas anteriormente.

```
1111static void kbd_keycode(unsigned int keycode, int down, int hw_raw)
1112{
1113        struct vc_data *vc = vc_cons[fg_console].d;
1114        unsigned short keysym, *key_map;
1115        unsigned char type, raw_mode;
1116        struct tty_struct *tty;
1117        int shift_final;
1118
1119        tty = vc->vc_tty;
1120
1121        if (tty && (!tty->driver_data)) {
1122                /* No driver data? Strange. Okay we fix it then. */
1123                tty->driver_data = vc;
1124        }
1125
1126        kbd = kbd_table + fg_console;
1127
1128        if (keycode == KEY_LEFTALT || keycode == KEY_RIGHTALT)
1129                sysrq_alt = down ? keycode : 0;
1130#ifdef CONFIG_SPARC
1131        if (keycode == KEY_STOP)
1132                sparc_l1_a_state = down;
1133#endif
1134
1135        rep = (down == 2);
1136
1137#ifdef CONFIG_MAC_EMUMOUSEBTN
1138        if (mac_hid_mouse_emulate_buttons(1, keycode, down))
1139                return;
1140#endif /* CONFIG_MAC_EMUMOUSEBTN */
1141
1142        if ((raw_mode = (kbd->kbdmode == VC_RAW)) && !hw_raw)
```

```
1143            if (emulate_raw(vc, keycode, !down << 7))
1144                    if (keycode < BTN_MISC && printk_ratelimit())
1145                            printk(KERN_WARNING "keyboard.c: can't emulate rawmode for keycode
%d\n", keycode);
1146
1147#ifdef CONFIG_MAGIC_SYSRQ              /* Handle the SysRq Hack */
1148        if (keycode == KEY_SYSRQ && (sysrq_down || (down == 1 && sysrq_alt))) {
1149            if (!sysrq_down) {
1150                    sysrq_down = down;
1151                    sysrq_alt_use = sysrq_alt;
1152            }
1153            return;
1154        }
1155        if (sysrq_down && !down && keycode == sysrq_alt_use)
1156            sysrq_down = 0;
1157        if (sysrq_down && down && !rep) {
1158            handle_sysrq(kbd_sysrq_xlate[keycode], tty);
1159            return;
1160        }
1161#endif
1162#ifdef CONFIG_SPARC
1163        if (keycode == KEY_A && sparc_l1_a_state) {
1164            sparc_l1_a_state = 0;
1165            sun_do_break();
1166        }
1167#endif
1168
1169        if (kbd->kbdmode == VC_MEDIUMRAW) {
1170            /*
1171             * This is extended medium raw mode, with keys above 127
1172             * encoded as 0, high 7 bits, low 7 bits, with the 0 bearing
1173             * the 'up' flag if needed. 0 is reserved, so this shouldn't
1174             * interfere with anything else. The two bytes after 0 will
1175             * always have the up flag set not to interfere with older
1176             * applications. This allows for 16384 different keycodes,
1177             * which should be enough.
1178             */
1179            if (keycode < 128) {
1180                    put_queue(vc, keycode | (!down << 7));
1181            } else {
1182                    put_queue(vc, !down << 7);
1183                    put_queue(vc, (keycode >> 7) | 0x80);
1184                    put_queue(vc, keycode | 0x80);
1185            }
1186            raw_mode = 1;
1187        }
1188
1189        if (down)
1190            set_bit(keycode, key_down);
1191        else
1192            clear_bit(keycode, key_down);
1193
1194        if (rep &&
1195           (!vc_kbd_mode(kbd, VC_REPEAT) ||
1196            (tty && !L_ECHO(tty) && tty->driver->chars_in_buffer(tty)))) {
1197            /*
1198             * Don't repeat a key if the input buffers are not empty and the
1199             * characters get aren't echoed locally. This makes key repeat
```

```
1200             * usable with slow applications and under heavy loads.
1201             */
1202            return;
1203    }
1204
1205    shift_final = (shift_state | kbd->slockstate) ^ kbd->lockstate;
1206    key_map = key_maps[shift_final];
1207
1208    if (!key_map) {
1209            compute_shiftstate();
1210            kbd->slockstate = 0;
1211            return;
1212    }
1213
1214    if (keycode > NR_KEYS)
1215            if (keycode >= KEY_BRL_DOT1 && keycode <= KEY_BRL_DOT8)
1216                    keysym = K(KT_BRL, keycode - KEY_BRL_DOT1 + 1);
1217            else
1218                    return;
1219    else
1220            keysym = key_map[keycode];
1221
1222    type = KTYP(keysym);
1223
1224    if (type < 0xf0) {
1225            if (down && !raw_mode)
1226                    to_utf8(vc, keysym);
1227            return;
1228    }
1229
1230    type -= 0xf0;
1231
1232    if (raw_mode && type != KT_SPEC && type != KT_SHIFT)
1233            return;
1234
1235    if (type == KT_LETTER) {
1236            type = KT_LATIN;
1237            if (vc_kbd_led(kbd, VC_CAPSLOCK)) {
1238                    key_map = key_maps[shift_final ^ (1 << KG_SHIFT)];
1239                    if (key_map)
1240                            keysym = key_map[keycode];
1241            }
1242    }
1243
1244    (*k_handler[type])(vc, keysym & 0xff, !down);
1245
1246    if (type != KT_SLOCK)
1247            kbd->slockstate = 0;
1248}
```

__kbd_init

Especifica la misma configuración de teclado para todas las consolas presentes en el sistema.

```
1357int __init kbd_init(void)
```

```
1358{
1359        int i;
1360        int error;
1361
1362        for (i = 0; i < MAX_NR_CONSOLES; i++) {
1363                kbd_table[i].ledflagstate = KBD_DEFLEDS;
1364                kbd_table[i].default_ledflagstate = KBD_DEFLEDS;
1365                kbd_table[i].ledmode = LED_SHOW_FLAGS;
1366                kbd_table[i].lockstate = KBD_DEFLOCK;
1367                kbd_table[i].slockstate = 0;
1368                kbd_table[i].modeflags = KBD_DEFMODE;
1369                kbd_table[i].kbdmode = VC_XLATE;
1370        }
1371
1372        error = input_register_handler(&kbd_handler);
1373        if (error)
1374                return error;
1375
1376        tasklet_enable(&keyboard_tasklet);
1377        tasklet_schedule(&keyboard_tasklet);
1378
1379        return 0;
1380}
```

tty_read

Función que implementa la lectura desde los dispositivos tty.

```
1720/**
1721 *      tty_read        -       read method for tty device files
1722 *      @file: pointer to tty file
1723 *      @buf: user buffer
1724 *      @count: size of user buffer
1725 *      @ppos: unused
1726 *
1727 *      Perform the read system call function on this terminal device. Checks
1728 *      for hung up devices before calling the line discipline method.
1729 *
1730 *      Locking:
1731 *              Locks the line discipline internally while needed
```

```
1732 *              For historical reasons the line discipline read method is
1733 *      invoked under the BKL. This will go away in time so do not rely on it
1734 *      in new code. Multiple read calls may be outstanding in parallel.
1735 */
1736
1737static ssize_t tty_read(struct file * file, char __user * buf, size_t count,
1738                         loff_t *ppos)
1739{
1740        int i;
1741        struct tty_struct * tty;
1742        struct inode *inode;
1743        struct tty_ldisc *ld;
1744
1745        tty = (struct tty_struct *)file->private_data;
1746        inode = file->f_path.dentry->d_inode;
1747        if (tty_paranoia_check(tty, inode, "tty_read"))
1748                return -EIO;
1749        if (!tty || (test_bit(TTY_IO_ERROR, &tty->flags)))
1750                return -EIO;
1751
1752        /* We want to wait for the line discipline to sort out in this
1753           situation */
1754        ld = tty_ldisc_ref_wait(tty);
1755        lock_kernel();
1756        if (ld->read)
1757                i = (ld->read)(tty,file,buf,count);
1758        else
1759                i = -EIO;
1760        tty_ldisc_deref(ld);
1761        unlock_kernel();
1762        if (i > 0)
1763                inode->i_atime = current_fs_time(inode->i_sb);
1764        return i;
1765}
```