LECCIÓN: IMPRESORA

IMPRESORA.1 Introducción

Como ya hemos visto anteriormente, el acceso a cualquier dispositivo se realiza a través de una serie de archivos especiales relacionados con el controlador del dispositivo. Existen dos tipos:

En modo bloque: correspondientes a dispositivos estructurados en bloque, a los que se accede proporcionando el número de bloque a leer o a escribir:

En modo carácter: Correspondientes a dispositivos no estructurados, a los que se accede leyendo o escribiendo byte a byte, generalmente de forma secuencial.

En este tema nos centraremos en un ejemplo de los dispositivos en modo carácter: las impresoras.

El sistema operativo accede al controlador de la impresora en **/dev/lpX** (siendo X 0, 1, 2,3…) que son enlaces simbolicos a las impresoras de la máquina.

Posee tres características comunes a cualquier controlador:

1.- Identificador del controlador.
2.- Identificador del dispositivo.
3.- Modo de funcionamiento: (Modo carácter en nuestro caso)

Este tipo de dispositivos posee la particularidad de ofrecer dos técnicas para comunicarse con el puerto:

- *Interrupciones*: Para señalar un evento, el hardware del controlador envía una interrupción que es tratada por la función manejadora, mediante las IRQ, este método libera trabajo del procesador.

- *Exploración o consulta (polling)*: en lugar de utilizar interrupciones, el manejador efectúa bucles de espera comprobando el puerto registro de estado del controlador. Este método permite evitar la producción de numerosas interrupciones (cada vez que se envíe un carácter a imprimir) y puede resultar más eficaz.

Configuración

El dispositivo debe estar conectado con el puerto paralelo.

Se puede configurar el driver mediante un comando en el núcleo.

```
58  *     # insmod lp.o parport=1,none,2
59  *
60  *     # insmod lp.o parport=auto
61  *
62  *     # insmod lp.o reset=1
```

El comando insmod es la instrucción utilizada por el programa modprobe para la carga de módulos en el sistema

- Lp=parpot1, none, parport2 ⇨ Puerto paralelo en el que está conectada.
- Lp=auto ⇨ Autodetectar impresoras.
- Lp=reset ⇨ Resetea la impresora durante la inicialización.
- Lp=off ⇨ Desactivar el driver de impresora completamente.

Para núcleos inferiores a 2.0, se le comunica la dirección I/O:

- Lp0 0x3bc
- Lp1 0x378
- Lp2 0x278

El manejador, por defecto, enlaza los dispositivos del lp a los dispositivos parport actuando éstos en su lugar. Esto significa que si se tiene solamente un puerto, estará limitado a lp0 sin importar su dirección de entrada-salida. Si se necesita el viejo comportamiento, se puede forzar usando los parámetros descritos arriba.

IMPRESORA 2. Estructuras de Datos

Estructura del puerto paralelo

```
225 struct pardevice {
226         const char *name;
227         struct parport *port;
228         int daisy;
229         int (*preempt)(void *);
230         void (*wakeup)(void *);
231         void *private;
232         void (*irq_func)(int, void *);
233         unsigned int flags;
234         struct pardevice *next;
235         struct pardevice *prev;
236         struct parport_state *state;     /* saved status over preemption */
237         wait_queue_head_t wait_q;
238         unsigned long int time;
239         unsigned long int timeslice;
```

```
240       volatile long int timeout;
241       unsigned long waiting;        /* long required for set_bit --RR */
242       struct pardevice *waitprev;
243       struct pardevice *waitnext;
244       void * sysctl_table;
245 };
```

Los puertos paralelos gestionados se definen por descriptores, agrupados en la tabla **lp_table.** La constante **LP_NO** (#define LP_NO 8) define el tamaño de la tabla (normalmente 8). Si se dispone de más de 8 impresoras es necesario incrementar este valor.

La estructura **lp_struct** (Struct lp_struct lp_table[LP_NO]) especifica el tipo de cada uno de los elementos y cada posición posee información de una impresora.

En resumen, **lp_table** corresponde a un vector de tamaño LP_NO de **lp_structs**, en los que se guardará información relativa a cada impresora.

*lp_struct **(definida en** lp.h**)***

```
134 struct lp_struct {
135       struct pardevice *dev;
136       unsigned long flags;
137       unsigned int chars;
138       unsigned int time;
139       unsigned int wait;
140       char *lp_buffer;
141 #ifdef LP_STATS
142       unsigned int lastcall;
143       unsigned int runchars;
144       struct lp_stats stats;
145 #endif
146       wait_queue_head_t waitq;
147       unsigned int last_error;
148       struct semaphore port_mutex;
149       wait_queue_head_t dataq;
150       long timeout;
151       unsigned int best_mode;
152       unsigned int current_mode;
153       unsigned long bits;
154 };
```

Tipo	*Campo*	*Descripción*
*Struct pardevice**	*dev*	*Estructura que guarda información asociada al dispositivo.*
Int	*Flags*	*Estado de la impresora conectada.*
Unsigned int	*Chars*	*Número de intentos a efectuar para imprimir un carácter.*
Unsigned int	*Time*	*Duración de la suspensión para una espera expresada en ciclos de reloj.*
Unsigned int	*Wait*	*Número de bucles de espera (de polling) a efectuar antes de que la impresora tenga en cuenta un carácter.*
*Char **	*Lp_buffer*	*Puntero a una memoria intermedia que contiene los caracteres a imprimir.*

Unsigned int	*Lastcall*	*Fecha de la última escritura en la impresora.*
Unsigned int	*Runchars*	*Número de caracteres escritos en la impresora sin provocar suspensión.*
Struct lp_stats	*Stats*	*Estadísticas sobre el uso de la impresora.*
*Struct *wait_queue_head_t*	*wait_q*	*Cola de espera utilizada para esperar la llegada de una interrupción.*
unsigned int	*last_error;*	*Almacena el último error producido en la impresora.*
Struct semaphore	*port_mutex*	*Semáforo para bloquear el puerto.*
wait_queue_head_t	*dataq*	*Cola de espera utilizada para esperar la llegada de los datos.*
long	*timeout*	*Tiempo de espera.*
unsigned int	*best_mode*	*Código del mejor modo negociado.*
unsigned int	*current_mode*	*Código del modo actual negociado.*
unsigned long	*bits*	*Bits que definen el estado del puerto asociado (testeo y limpieza).*

lp_fops ***(definida en** lp.c**)***

Contiene las operaciones que se van a realizar sobre la impresora, inicializando cada campo a las funciones que permiten operar sobre la misma.

```
662 static struct file_operations lp_fops = {
663         .owner          = THIS_MODULE,
664         .write          = lp_write,
665         .ioctl          = lp_ioctl,
666         .open           = lp_open,
667         .release        = lp_release,
668 #ifdef CONFIG_PARPORT_1284
669         .read           = lp_read,
670 #endif
 671 };
```

Campo	*Descripción*
.owner	Obtener información sobre el propio módulo.
.write	Operación que se encarga de realizar el envío de los datos a la impresora.
.ioctl	Operación que permite consultar y modificar parámetros relacionados con la impresora.
.open	Operación que permite realizar la apertura del archivo sobre la impresora.
.release	Operación que permite dejar la impresora desocupada y libre para usar.
.read	Operación que se encarga de realizar la obtención de los datos a la impresora.

IMPRESORA 3. Procedimientos

Operaciones sobre los puertos:

lp_claim_parport_or_block

Este procedimiento se usa para demandar el puerto paralelo o bloque a menos que ya sea nuestro.

```
168 static void lp_claim_parport_or_block(struct lp_struct *this_lp)
169 {
170         if (!test_and_set_bit(LP_PARPORT_CLAIMED, &this_lp->bits)) {
171                 parport_claim_or_block (this_lp->dev);
172         }
173 }
```

Parport_claim_or_block es una función definida en /linux/drivers/parport/share.c que adquiere en exclusiva el puerto paralelo o pone a dormir el proceso en wait_q en espera de una señal de la impresora.

lp_release_parport

Este procedimiento se usa para liberar el puerto que ha sido previamente tomado por la función anterior.

```
176 static void lp_release_parport(struct lp_struct *this_lp)
177 {
178         if (test_and_clear_bit(LP_PARPORT_CLAIMED, &this_lp->bits)) {
179                 parport_release (this_lp->dev);
180         }
181 }
```

Parport_release es otra función de /linux/drivers/parport/share.c que termina el acceso a la impresora. Siempre devuelve una salida positiva.

lp_preempt

Función para dar prioridad a un dispositivo, activando el bit LP_PREEMPT_REQUEST.

```
185 static int lp_preempt(void *handle)
186 {
187         struct lp_struct *this_lp = (struct lp_struct *)handle;
188         set_bit(LP_PREEMPT_REQUEST, &this_lp->bits);
189         return (1);
190 }
```

lp_negotiate

Esta función se usa para negociar con el puerto el modo de transferencia. Si no se consigue, asigna el modo compatible de forma predeterminada. La función parport_negociate establece el modo de transferencia que debe ser necesariamente

uno de los establecidos por IEEE. Si consigue establecer el modo con éxito devolverá 0, si no cumple las especificaciones IEEE devuelve
-1 y si el dispositivo rechaza la conexión devuelve 1.

```
197 static int lp_negotiate(struct parport * port, int mode)
198 {
        //Se negocia el modo de operación del puerto pasado por parámetro, si no lo consigue
        //se  asigna el modo compat.
199     if (parport_negotiate (port, mode) != 0) {
200         mode = IEEE1284_MODE_COMPAT;
201         parport_negotiate (port, mode);
202     }
203
204     return (mode);
205 }
```

Los modos de transferencia para puerto paralelo definidos en el estándar IEEE 1284 son:

- **Modo de Compatibilidad** (modo estándar o “Centronics”)
- **Modo Nibble**: 4 bits a la vez usando las líneas de estado (Status) para datos (Hewlett Packard Bi-tronics)
- **Modo de Octeto (Byte Mode)**: 8 bits a la vez usando las líneas de datos, a veces nombrado como puerto bidireccional
- **EPP (Enhanced Parallel Port)**: Puerto Paralelo Extendido, usado principalmente para periféricos que no son impresoras, como CD-ROM, Adaptadores de Red, etc.
- **ECP (Extended Capability Port)**: Puerto de Capacidades Extendidas, usado principalmente por impresoras recientes y scanners.

lp_reset

Resetea la impresora. La macro w_ctr se encarga de enviar las señales a través del puerto para reiniciar la impresora. Con la macro r_str leemos el estado actual del dispositivo.

```
159 #define r_str(x)        (parport_read_status(lp_table[(x)].dev->port))
160 #define w_ctr(x,y)      do { parport_write_control(lp_table[(x)].dev->port, (y)); } while (0)

207 static int lp_reset(int minor)
208 {
209     int retval;
        //Solicita el puerto paralelo o bloque a menos de que esté ocupado.
210     lp_claim_parport_or_block (&lp_table[minor]);
        //Reinicia
211     w_ctr(minor, LP_PSELECP);
212     udelay (LP_DELAY);
213     w_ctr(minor, LP_PSELECP | LP_PINITP);
214     retval = r_str(minor);
        //Libera el puerto:
215     lp_release_parport (&lp_table[minor]);
216     return retval;
217 }
```

lp_error

Procedimiento invocado en caso de error. En caso de que el dispositivo esté llevando a cabo una tarea que no se puede interrumpir, pone la tarea en espera mientras el dispositivo se recupera del error.

```
219 static void lp_error (int minor)
220 {
221         DEFINE_WAIT(wait);
222         int polling;
223 :
224         if (LP_F(minor) & LP_ABORT)
225                 return;
226
                  //Si estamos en modo consulta liberamos el puerto
227         polling = lp_table[minor].dev->port->irq == PARPORT_IRQ_NONE;
228         if (polling) lp_release_parport (&lp_table[minor]);

                  //Se pone en espera al dispositivo
229         prepare_to_wait(&lp_table[minor].waitq, &wait, TASK_INTERRUPTIBLE);
230         schedule_timeout(LP_TIMEOUT_POLLED);
231         finish_wait(&lp_table[minor].waitq, &wait);

                  //Si estaba en modo consulta reclama el puerto, si no, lo libera temporalmente
232         if (polling) lp_claim_parport_or_block (&lp_table[minor]);
233         else parport_yield_blocking (lp_table[minor].dev);
234 }
```

lp_check_status

Consulta para averiguar el estado de la impresora.

```
234 static int lp_check_status(int minor)
235 {       //Inicializaciones:
236         int error = 0;
237         unsigned int last = lp_table[minor].last_error;
238         unsigned char status = r_str(minor);
239         if ((status & LP_PERRORP) && !(LP_F(minor) & LP_CAREFUL))
240                  //No hay error
241                 last = 0;
                  //Si hay errores, se comprueba que tipo de error hay y se marca en error el
                  //tipo correspondiente.
                  //FALTA DE PAPEL:
242         else if ((status & LP_POUTPA)) {
243                 if (last != LP_POUTPA) {
244                         last = LP_POUTPA;
245                         printk(KERN_INFO "lp%d out of paper\n", minor);
246                 }
247                 error = -ENOSPC;
                 //DESCONECTADA:
248         } else if (!(status & LP_PSELECD)) {
249                 if (last != LP_PSELECD) {
```

```
250                     last = LP_PSELECD;
251                     printk(KERN_INFO "lp%d off-line\n", minor);
252             }
253             error = -EIO;
                //OTRO TIPO DE ERROR:
254     } else if (!(status & LP_PERRORP)) {
255             if (last != LP_PERRORP) {
256                     last = LP_PERRORP;
257                     printk(KERN_INFO "lp%d on fire\n", minor);
258             }
259             error = -EIO;
260     } else {
                //Si LP_CAREFUL está activado y no se han encontrado errores:
261             last = 0;
263     }
264      //Se almacena el tipo de error producido:
265     lp_table[minor].last_error = last;
266     //Se llama a  lp_error para que muestre el mensaje de error producido:
267     if (last != 0)
268             lp_error(minor);
270     return error;
271 }
```

Operaciones de impresora:

lp_write

¿Qué hace lp write?

Lp_write se ocupa de enviar los datos de usuario a la impresora. Para ello, estos datos se dividen en bloques, cuyo tamaño no podrá exceder el del buffer.

La función toma el puerto paralelo y envía los datos, mientras hayan datos para enviar y no existan errores de alguna clase. Al finalizar el envío de datos, liberará el puerto.

¿Qué parámetros utiliza?

struct file *file -> Estructura relacionada con el fichero para el cual se llamó esta rutina.
const char __user * buf -> Este puntero apunta a los datos de usuario a enviar.
size_t count -> La longitud de estos datos.
loff_t *ppos -> Sin uso aparente desde largo tiempo atrás.

¿Cómo lo hace? – Código de la función:

En primer lugar, la función debe identificar el dispositivo que usará. Éste está indicado como el dispositivo de menor número dentro del inodo incluido en la estructura de archivo que se le ha pasado por parámetro.

Además de esto, se realizan inicializaciones varias que serán utilizadas a lo largo del código de la función.

```
292static ssize_t lp_write(struct file * file, const char __user * buf,
293                         size_t count, loff_t *ppos)
294{     /* Inicializando algunos datos para su uso más adelante */
295      unsigned int minor = iminor(file->f_path.dentry->d_inode);
 // La línea anterior encontrará  el dispositivo de menor número al que apunta el inodo del archivo.
296        struct parport *port = lp_table[minor].dev->port;
297        char *kbuf = lp_table[minor].lp_buffer;
298        ssize_t retv = 0;
299        ssize_t written;
300        size_t copy_size = count;
301        int nonblock = ((file->f_flags & O_NONBLOCK) ||
302                        (LP_F(minor) & LP_ABORT));
303
304#ifdef LP_STATS
305        if (time_after(jiffies, lp_table[minor].lastcall + LP_TIME(minor)))
306                lp_table[minor].runchars = 0;
307
308        lp_table[minor].lastcall = jiffies;
309#endif
```

Llegado este punto es cuando comienza el código “real” de la función. En primer lugar, controlará que el tamaño de la copia sea menor o igual que el tamaño del buffer del dispositivo. Tras esto, tomará el cerrojo, copiará los datos de usuario al buffer (en el código, el puntero kbuf) y reclamará el puerto paralelo para su uso.

En caso de que el puerto no estuviera disponible, se bloqueará hasta que lo esté...

```
311       /* Controlamos que sólo se copia un tamaño que quepa en el buffer */
 312       if (copy_size > LP_BUFFER_SIZE)
 313               copy_size = LP_BUFFER_SIZE;
 314  /* Cerrojo */
 315       if (mutex_lock_interruptible(&lp_table[minor].port_mutex))
 316               return -EINTR;
 317 /* Copiamos los datos del usuario */
 318       if (copy_from_user (kbuf, buf, copy_size)) {
 319               retv = -EFAULT;
 320               goto out_unlock;
 321       }
 323      /* Reclamar Parport o dormir hasta que esté disponible
 324       */
 325       lp_claim_parport_or_block (&lp_table[minor]);
```

Tras esto, el modo actual del dispositivo se negociará para que sea el mejor modo posible. El mejor modo posible fue definido en la llamada lp_open() anterior a esta llamada a write y generalmente es el modo ECP, salvo que no sea aceptado, en cuyo caso es el modo de compatibilidad.

Una vez hecho esto, comienza el envío de datos, el cual se hace en un bucle hasta que dejen de haber datos que enviar o bien ocurra un error.

Este error se puede manifestar en forma de un bloque no enviado por entero, así que cuando esto sucede se comprobará si la impresora sigue lista para continuar el envío o bien ya no lo está y efectivamente hay un error.

```
326  /* El modo actual ha de ser el más apropiado..... */
 327       lp_table[minor].current_mode = lp_negotiate (port,
 328                                               lp_table[minor].best_mode);
330       parport_set_timeout (lp_table[minor].dev,
 331                            (nonblock ? PARPORT_INACTIVITY_O_NONBLOCK
 332                             : lp_table[minor].timeout));
 333       // Mientras el tamaño de los datos por enviar sea mayor que cero, vamos a escribir
 334       if ((retv = lp_wait_ready (minor, nonblock)) == 0)
 335       do {
 336               /* Pasando  los datos a través del puerto */
 337               written = parport_write (port, kbuf, copy_size);
 338               if (written > 0) {
 339                       copy_size -= written;
 340                       count -= written;
 341                       buf  += written;
 342                       retv += written;
 343               }
345               if (signal_pending (current)) {
 346                       if (retv == 0)
 347                               retv = -EINTR;
349                       break;
 350           }
352               if (copy_size > 0) {
```

```
 353                    /* Si la escritura no se completó, puede haber un fallo, buscamos el causante */
 354                    int error;
356                    parport_negotiate (lp_table[minor].dev->port,
 357                                       IEEE1284_MODE_COMPAT);
 358                    lp_table[minor].current_mode = IEEE1284_MODE_COMPAT;
 359
 360                    error = lp_wait_ready (minor, nonblock);
 361                  /* Caso de error salimos del bucle */
 362                    if (error) {
 363                          if (retv == 0)
 364                                retv = error;
 365                          break;
 366                    } else if (nonblock) {
 367                          if (retv == 0)
 368                                retv = -EAGAIN;
 369                          break;
 370                    }
// Si no fue un error, restablecemos el modo más apropiado para el dispositivo.
372                    parport_yield_blocking (lp_table[minor].dev);
 373                    lp_table[minor].current_mode
 374                      = lp_negotiate (port,
 375                                      lp_table[minor].best_mode);
377              } else if (need_resched())
 378                    schedule ();
 379              // Si quedaran datos por escribir, se vuelve a rellenar el buffer de igual forma que antes
 380               if (count) {
 381                    copy_size = count;
 382                    if (copy_size > LP_BUFFER_SIZE)
 383                          copy_size = LP_BUFFER_SIZE;
385                   if (copy_from_user(kbuf, buf, copy_size)) {
 386                          if (retv == 0)
 387                                retv = -EFAULT;
 388                          break;
 389                    }
 390               }
 391        } while (count > 0);
```

Finalmente, tras acabar el envío de datos, se libera el bit de prioridad del dispositivo, se negocia el modo actual del dispositivo a modo de compatibilidad y se liberan tanto el puerto como el cerrojo tomado al inicio de la función, con lo que finaliza la función.

El dato devuelto en caso de que todo haya acabado correctamente es un 0.

```
/* Hora de salir: comprobamos el estado , volvemos a cambiar el modo del dispositivo y liberamos el
puerto paralelo */
393        if (test_and_clear_bit(LP_PREEMPT_REQUEST,
 394                               &lp_table[minor].bits)) {
 395                printk(KERN_INFO "lp%d releasing parport\n", minor);
 396                parport_negotiate (lp_table[minor].dev->port,
 397                                   IEEE1284_MODE_COMPAT);
 398                lp_table[minor].current_mode = IEEE1284_MODE_COMPAT;
 399                lp_release_parport (&lp_table[minor]);
 400        }
 401out_unlock:
// y cómo no, liberar el cerrojo antes de salir...
 402        mutex_unlock(&lp_table[minor].port_mutex);
 403
 404        return retv;
 405}
```

lp_ioctl

¿Qué hace lp ioctl?

Lp_ioctl nos da la posibilidad de consultar los distintos parámetros de una impresora y de modificar aquellos que pueden ser cambiados al gusto del usuario.

¿Qué parámetros utiliza?

struct inode ***inode** -> El inodo correspondiente al dispositivo.
struct file ***file** -> Estructura relacionada con el fichero para el cual se llamó esta rutina.
unsigned int cmd -> Indica el parámetro que consultaremos o modificaremos.
unsigned long arg -> Permite indicar el valor al que cambiaremos el parámetro.

¿Cómo lo hace? – Código de la función:

En primer lugar, la función debe identificar el dispositivo que usará. Éste está indicado como el dispositivo de menor número dentro del inodo que a tal efecto se le pasará por parámetro.

Además de esto, se realizan inicializaciones varias que serán utilizadas a lo largo del código de la función.

```
560static int lp_ioctl(struct inode *inode, struct file *file,
 561                     unsigned int cmd, unsigned long arg)
 562{
 563        unsigned int minor = iminor(inode);
 564        int status;
 565        int retval = 0;
 566        void __user *argp = (void __user *)arg;
 567
 568#ifdef LP_DEBUG
 569        printk(KERN_DEBUG "lp%d ioctl, cmd: 0x%x, arg: 0x%lx\n", minor, cmd, arg);
 570#endif
```

Ahora controlamos que el dispositivo esté en el rango de impresoras y que la impresora elegida realmente esté,en caso contrario se retornará el error oportuno.

```
571         if (minor >= LP_NO)
 572                 return -ENODEV;
 573         if ((LP_F(minor) & LP_EXIST) == 0)
 574                 return -ENODEV;
 575         switch ( cmd ) {
 576                 struct timeval par_timeout;
 577                 long to_jiffies;
```

En este momento se empiezan a tratar los distintos parámetros disponibles para consulta o modificación.
Los primeros cuatro casos son LP_TIME, que puede ser modificado para indicar la cantidad de tiempo que el driver esperará por el dispositivo, LP_CHAR permite

cambiar el tiempo a esperar por un carácter, LP_ABORT permite que se responda a un error reintentando (pasando un 0) o abortando (cualquier otro valor) y LP_ABORTOPEN hace exactamente lo mismo pero para la función lp_open() que veremos más adelante:

```
579             case LPTIME: // Fija cantidad de tiempo que el driver esperará por el dispositivo.
 580                     LP_TIME(minor) = arg * HZ/100;
 581                     break;
 582             case LPCHAR: // Fija cantidad de tiempo máxima para cada carácter
 583                     LP_CHAR(minor) = arg;
 584                     break;
 585             case LPABORT:
 586                     if (arg) // Si el argumento no es 0, entonces causa un error, si no, causa un
reintento
 587                             LP_F(minor) |= LP_ABORT;
 588                     else
 589                             LP_F(minor) &= ~LP_ABORT;
 590                     break;
 591             case LPABORTOPEN:
 592                     if (arg) // Si arg==0, se ignorarán errores en open(), si no, se abortará si hubieran
errores.
 593                             LP_F(minor) |= LP_ABORTOPEN;
 594                     else
 595                             LP_F(minor) &= ~LP_ABORTOPEN;
 596                     break;
```

Los siguientes cuatro casos son LP_CAREFUL, que al escribir ignorará los flag (valor 0) o bien los comprobará todos (resto de valores), LP_WAIT, que permite modificar el tiempo que el driver espera antes y después de un “strobe”, lo cual según lo que hemos encontrado es una ráfaga emitida por una señal especial llamada “de estroboscopio” que funciona mediante ráfagas, LP_SETIRQ devuelve directamente un error (hace pensar que en versiones antiguas se podía fijar la línea de interrupción y ahora no) y LP_GETIRQ devuelve al usuario el número de la línea de interrupción:

```
597             case LPCAREFUL:
 598                     if (arg) // Si no es 0, el driver se asegurará de que todo flag sea comprobado al
escribir.
 599                             LP_F(minor) |= LP_CAREFUL;
 600                     else // y si lo es, pues se ignora.

601                             LP_F(minor) &= ~LP_CAREFUL;
 602                     break;
 603             case LPWAIT: // Cantidad de tiempo que el driver espera antes y después de un
“strobe”
 604                     LP_WAIT(minor) = arg;
 605                     break;
 606             case LPSETIRQ:
 607                     return -EINVAL;
 608                     break;
 609             case LPGETIRQ: // Obtiene el número de IRQ.
 610                     if (copy_to_user(argp, &LP_IRQ(minor),
 611                                     sizeof(int)))
 612                             return -EFAULT;
 613                     break;
```

Los siguientes casos son LPGETSTATUS, que consulta el estado de la impresora mediante r_str y le copia los resultados a usuario; LP_RESET, que llamará a la función de reset explicada anteriormente y LP_GETSTATS que devuelve al usuario las estadísticas de la impresora, siempre en caso de que éstas estén disponibles:

```
614             case LPGETSTATUS:// Obtendrá el status de la impresora
 615                     lp_claim_parport_or_block (&lp_table[minor]);
 616                     status = r_str(minor);
 617                     lp_release_parport (&lp_table[minor]);
 619                     if (copy_to_user(argp, &status, sizeof(int)))
 620                             return -EFAULT;
 621                     break;
 622              case LPRESET:  //Resetea la impresora
 623                     lp_reset(minor);
 624                     break;
 625#ifdef LP_STATS
 626              case LPGETSTATS: // Estadísticas actuales del driver
 627                     if (copy_to_user(argp, &LP_STAT(minor),
 628                                     sizeof(struct lp_stats)))
 629                             return -EFAULT;
 630                     if (capable(CAP_SYS_ADMIN))
 631                             memset(&LP_STAT(minor), 0,
 632                                             sizeof(struct lp_stats));
 633                     break;
 634#endif
```

Finalmente, los dos últimos casos son LPGETFLAGS, que devuelve al usuario una copia de los flags del dispositivo y LPSETTIMEOUT, la cual toma una estructura timeout pasada por el usuario, traduce los valores a la unidad de tiempo “jiffies” y actualiza el campo timeout de nuestro dispositivo con ella.

La función, si ha finalizado con éxito, devuelve 0. En caso de que el parámetro no se corresponda con ninguno de los comentados aquí o bien haya producido algún tipo de error, la función retorna un valor de error.

```
635             case LPGETFLAGS: // Obtiene los flags
 636                     status = LP_F(minor);
 637                     if (copy_to_user(argp, &status, sizeof(int)))
 638                             return -EFAULT;
 639                     break;
 641              case LPSETTIMEOUT: // Fijar el parámetro timeout en la estructura lpstruct del
dispositivo actual
 642                     if (copy_from_user (&par_timeout, argp,
 643                                     sizeof (struct timeval))) {
 644                             return -EFAULT;
 645                     }
 646     /* Y ahora convertir  a jiffies (unidad de tiempo)  y emplazarlo en lp_table */

647                     if ((par_timeout.tv_sec < 0) ||
 648                         (par_timeout.tv_usec < 0)) {
 649                             return -EINVAL;
 650                     }
 651                     to_jiffies = DIV_ROUND_UP(par_timeout.tv_usec, 1000000/HZ);
 652                     to_jiffies += par_timeout.tv_sec * (long) HZ;

653                     if (to_jiffies <= 0) {
 654                             return -EINVAL;
 655                     }
```

```
656                     lp_table[minor].timeout = to_jiffies;
657                     break;
658
659             default: // Si el caso no está contemplado, devolvemos -EINVAL

660                     retval = -EINVAL;
661     }
662     return retval;
663}
```

lp_open

¿Qué hace lp open?

Lp_open nos permite abrir una comunicación de datos con la impresora a través de un puerto paralelo.

Es imprescindible efectuar esta operación antes que cualquiera otra que se desee realizar, como por ejemplo un envío de datos para imprimir.

¿Qué parámetros utiliza?

struct inode ***inode** -> El inodo correspondiente al dispositivo
struct file ***file** -> Estructura relacionada con el fichero para el cual se llamó esta rutina.

¿Cómo lo hace? – Código de la función:

En primer lugar, la función debe identificar el dispositivo que usará. Éste está indicado como el dispositivo de menor número dentro del inodo que a tal efecto se le pasará por parámetro. Esto es el primer paso de todas las funciones de impresora.

Luego controlará que el dispositivo esté en el rango de impresoras, que la impresora esté ahí y que además no esté ocupada, en cuyo caso seguirá adelante. Si no fuera así, se retornaría un valor de error.

Además, en el caso de que se hubiera activado LP_ABORTOPEN en la función lp_ioctl, se controlarán errores como la ausencia de papel o que la impresora esté apagada.

```
489static int lp_open(struct inode * inode, struct file * file)
490{
491     unsigned int minor = iminor(inode);
492   /* Pilla el dispositivo menor y pregunta si está en el rango de impresoras, si la impresora existe
y si la impresora no está ocupada, devolviendo errores si fallan los test */
493     if (minor >= LP_NO)
494          return -ENXIO;
495     if ((LP_F(minor) & LP_EXIST) == 0)
496          return -ENXIO;
497     if (test_and_set_bit(LP_BUSY_BIT_POS, &LP_F(minor)))
```

```
498             return -EBUSY;
499  // Caso de que ABORTOPEN activo y existan errores tipo falta de papel, los controla
505       if ((LP_F(minor) & LP_ABORTOPEN) && !(file->f_flags & O_NONBLOCK)) {
506             int status;
507             lp_claim_parport_or_block (&lp_table[minor]);
508             status = r_str(minor);
509             lp_release_parport (&lp_table[minor]);
510             if (status & LP_POUTPA) {
511                   printk(KERN_INFO "lp%d out of paper\n", minor);
512                   LP_F(minor) &= ~LP_BUSY;
513                  return -ENOSPC;
514             } else if (!(status & LP_PSELECD)) {
515                   printk(KERN_INFO "lp%d off-line\n", minor);
516                   LP_F(minor) &= ~LP_BUSY;
517                   return -EIO;
518             } else if (!(status & LP_PERRORP)) {
519                   printk(KERN_ERR "lp%d printer error\n", minor);
520                   LP_F(minor) &= ~LP_BUSY;
521                   return -EIO;
522             }
523       }
```

Si no han habido problemas, creamos un espacio de memoria para el buffer de impresora y si se crea correctamente tomamos el puerto paralelo y negociamos el mejor modo para la impresora. Éste será el modo ECP, a menos que la impresora no lo soporte, en cuyo caso el modo será el modo de compatibilidad.

Finalmente, pondremos el modo actual de operación a modo de compatibilidad, pues todavía no se está trabajando con la impresora, y liberamos el puerto paralelo, tras lo cual se sale de la función.

Tanto la creación del buffer como la asignación de modos actualiza los campos correspondientes en el lp_struct correspondiente al dispositivo con el que estemos trabajando.

```
524       lp_table[minor].lp_buffer = kmalloc(LP_BUFFER_SIZE, GFP_KERNEL); /* Crea memoria
para el búfer. Si no hubiera memoria disponible, canta error y cierra */
525       if (!lp_table[minor].lp_buffer) {
526             LP_F(minor) &= ~LP_BUSY;
527             return -ENOMEM;
528       }
529       /* Comprueba si el periférico soporta el modo ECP y en tal caso lo establece como mejor
modo. Caso contrario, el mejor modo será el modo de compatibilidad de IEEE1284 */
530       lp_claim_parport_or_block (&lp_table[minor]);
531       if ( (lp_table[minor].dev->port->modes & PARPORT_MODE_ECP) &&
532            !parport_negotiate (lp_table[minor].dev->port,
533                                IEEE1284_MODE_ECP)) {
534             printk (KERN_INFO "lp%d: ECP mode\n", minor);
535             lp_table[minor].best_mode = IEEE1284_MODE_ECP;
536       } else {
537             lp_table[minor].best_mode = IEEE1284_MODE_COMPAT;
538       }
539       /* Deja el modo actual del periférico establecido en modo de compatibilidad y retorna*/
540       parport_negotiate (lp_table[minor].dev->port, IEEE1284_MODE_COMPAT);
541       lp_release_parport (&lp_table[minor]);
542       lp_table[minor].current_mode = IEEE1284_MODE_COMPAT;
543       return 0;
544}
```

lp_release

¿Qué hace lp release?

Lp_release nos permite cerrar la comunicación de datos con la impresora. Esta función es corta y es contraria de la función lp_open.

¿Qué parámetros utiliza?

struct inode *inode -> El inodo correspondiente al dispositivo
struct file *file -> El fichero para el cual se llamó esta rutina.

¿Cómo lo hace? – Código de la función:

Como en todas las funciones anteriores, el primer paso es identificar el dispositivo que usará. Éste está indicado como el dispositivo de menor número dentro del inodo que a tal efecto se le pasará por parámetro.

Tras esto, se tomará el puerto paralelo para negociar el modo de operación actual a modo de compatibilidad, y una vez hecho esto se vuelve a liberar.

Finalmente, se libera el espacio de memoria tomado en lp_open para el buffer del dispositivo, se asigna el puntero de buffer de la estructura lp_struct correspondiente a NULL y se marca a 0 el flag de ocupado de esa misma estructura.

La función finaliza retornando 0.

```
546static int lp_release(struct inode * inode, struct file * file)
 547{
 548       unsigned int minor = iminor(inode);
 549       /* Esta función pone el modo actual de la impresora en modo de compatibilidad, liberará el
espacio en memoria que fue adquirido para el búfer en la función lp_open() y finalmente cambia los
flags para marcar que la impresora ya no está ocupada */
 550       lp_claim_parport_or_block (&lp_table[minor]);
 551       parport_negotiate (lp_table[minor].dev->port, IEEE1284_MODE_COMPAT);
 552       lp_table[minor].current_mode = IEEE1284_MODE_COMPAT;
 553       lp_release_parport (&lp_table[minor]);
 554       kfree(lp_table[minor].lp_buffer);
 555       lp_table[minor].lp_buffer = NULL;
 556       LP_F(minor) &= ~LP_BUSY;
 557       return 0;
 558}
```

lp_read

¿Qué hace lp read?

Esta función tendrá como objetivo obtener los datos de la impresora.

De igual forma que en la función lp_write, es necesario leer en bloques que no superen el tamaño del buffer, de hecho, la estructura de las dos funciones es muy parecida.

La función toma el puerto paralelo y leerá los datos, mientras hayan datos para leer y no existan errores de alguna clase. Al finalizar la lectura de datos, liberará el puerto y copiará los datos al usuario guardándolos en el parámetro pasado a tal efecto.

¿Qué parámetros utiliza?

struct file ***file** -> Estructura relacionada con el fichero para el cual se llamó esta rutina.
const char __user ***** **buf** -> Puntero en el cual se copiarán los datos leídos.
size_t **count** -> La longitud de los datos.
loff_t *ppos -> Sin uso aparente desde largo tiempo atrás.

¿Cómo lo hace? – Código de la función:

Como en todas las funciones anteriores, el primer paso es identificar el dispositivo que usará. Éste está indicado como el dispositivo de menor número dentro del inodo incluido en la estructura de archivo que se le ha pasado por parámetro.

Además de esto, se realizan inicializaciones varias que serán utilizadas a lo largo del código de la función:

```
410static ssize_t lp_read(struct file * file, char __user * buf,
 411                       size_t count, loff_t *ppos)
 412{   /* Detección del dispositivo de trabajo  y algunas inicializaciones */
 413       DEFINE_WAIT(wait);
 414       unsigned int minor=iminor(file->f_path.dentry->d_inode);
 415       struct parport *port = lp_table[minor].dev->port;
 416       ssize_t retval = 0;
 417       char *kbuf = lp_table[minor].lp_buffer;
 418       int nonblock = ((file->f_flags & O_NONBLOCK) ||
 419                       (LP_F(minor) & LP_ABORT));
```

Llegado este punto comienza el código “real” de la función. En primer lugar, se comprobará que los bloques sean de un tamaño no superior al del buffer. Tras esto, se tomará el cerrojo y el puerto paralelo. Como es costumbre, si el puerto paralelo estuviera en uso esperamos a que se libere para cogerlo.

Finalmente negociamos el modo de operación a modo de compatibilidad. Si el aceptado en lugar de ese fuera el modo Nibble, estamos ante un error y salimos.

```
420 /* Nos aseguramos de copiar en trozos no mayores que el tamaño del buffer */
 421       if (count > LP_BUFFER_SIZE)
 422               count = LP_BUFFER_SIZE;
 423 /* Tomamos el control del cerrojo y del puerto paralelo, o bien dormimos hasta que el puerto esté
libre */
 424       if (mutex_lock_interruptible(&lp_table[minor].port_mutex))
 425               return -EINTR;
 427       lp_claim_parport_or_block (&lp_table[minor]);
```

```
429       parport_set_timeout (lp_table[minor].dev,
 430                            (nonblock ? PARPORT_INACTIVITY_O_NONBLOCK
 431                             : lp_table[minor].timeout));
 432      /* Negociamos el modo de operación del puerto a modo de compatibilidad. Si el modo
disponible fuera el NIBBLE, salimos */
 433       parport_negotiate (lp_table[minor].dev->port, IEEE1284_MODE_COMPAT);
 434       if (parport_negotiate (lp_table[minor].dev->port,
 435                              IEEE1284_MODE_NIBBLE)) {
 436             retval = -EIO;
 437             goto out;
 438       }
```

Si todo ha ido bien, comenzamos la lectura de datos, que al igual que la escritura en su momento se hace en un bucle del cual sólo se sale cuando ya no hay más datos que leer o estamos ante un error. Los datos leídos se irán guardando en el buffer del dispositivo.

Un posible error puede surgir en forma de que no haya líneas de interrupción para el puerto paralelo, en cuyo caso se llamará a la función de error explicada anteriormente.

De no haber errores, si hubiera que leer varias veces se esperará un cierto tiempo antes de volver a leer, esto se hará entrando en la cola de espera el tiempo indicado mediante LP_TIMEOUT_POLLING y regresando de ella pasado dicho tiempo.

```
439       // Leemos los datos...
 440        while (retval == 0) {
 441             retval = parport_read (port, kbuf, count);
 443             if (retval > 0)
 444                  break;
446             if (nonblock) {
 447                  retval = -EAGAIN;
 448                  break;
 449             }
452 /* Controlamos posibles errores y esperamos por el siguiente dato a leer */
 453             if (lp_table[minor].dev->port->irq == PARPORT_IRQ_NONE) {
 454                  parport_negotiate (lp_table[minor].dev->port,
 455                                     IEEE1284_MODE_COMPAT);
 456                  lp_error (minor);
 457                  if (parport_negotiate (lp_table[minor].dev->port,
 458                                         IEEE1284_MODE_NIBBLE)) {
 459                       retval = -EIO;
 460                       goto out;
 461                  }
 462             } else {
 463                  prepare_to_wait(&lp_table[minor].waitq, &wait, TASK_INTERRUPTIBLE);
 464                  schedule_timeout(LP_TIMEOUT_POLLED); //Esperando por el dato mediante
polling.
 465                  finish_wait(&lp_table[minor].waitq, &wait);
 466             }
468             if (signal_pending (current)) {
 469                  retval = -ERESTARTSYS;
 470                  break;
 471             }
 473             cond_resched ();
 474       }
```

Finalmente, al finalizar la lectura de datos renegociamos el modo de operación a modo por compatibilidad, liberamos el puerto paralelo, copiamos los datos leídos al usuario y liberamos el cerrojo. La función retornará cero en caso de que todo haya ido correctamente.

```
/* Al terminar la lectura de datos, renegociamos el modo de operación y liberamos el puerto */
 475        parport_negotiate (lp_table[minor].dev->port, IEEE1284_MODE_COMPAT);
 476 out:
 477        lp_release_parport (&lp_table[minor]);
 478 /* Copiamos los datos leídos para que el usuario pueda disponer de ellos */
 479        if (retval > 0 && copy_to_user (buf, kbuf, retval))
 480                retval = -EFAULT;
 481 /* y liberamos el cerrojo antes de retornar*/
 482        mutex_unlock(&lp_table[minor].port_mutex);
 483
 484        return retval;
 485}
```

lp_init

¿Qué hace lp init?

Esta función inicializa los parámetros del manejador y es llamada únicamente cuando se inicia el sistema o bien cuando se cargue el gestor en forma de módulo.

El usuario no tiene la opción de utilizar esta función.

¿Qué parámetros utiliza?

Esta función no utiliza parámetros.

¿Cómo lo hace? – Código de la función:

En primer lugar, comprueba que el puerto esté activo e inicializa una a una las estructuras lp_struct existentes en lp_table dando valores por defecto a sus campos.

En lp_table hay LP_NO estructuras para rellenar, por defecto este parámetro está a 8, pudiendo cambiarse si hay más impresoras.

```
869static int __init lp_init (void)
 870{
 871        int i, err = 0;
 872   /* Miramos que el puerto esté activo*/
 873        if (parport_nr[0] == LP_PARPORT_OFF)
 874                return 0;
 875 /* Inicializamos una a una las LP_NO estructuras lp_struct que existan (por defecto 8) */
 876        for (i = 0; i < LP_NO; i++) {
 877                lp_table[i].dev = NULL;
 878                lp_table[i].flags = 0;
 879                lp_table[i].chars = LP_INIT_CHAR;
 880                lp_table[i].time = LP_INIT_TIME;
 881                lp_table[i].wait = LP_INIT_WAIT;
 882                lp_table[i].lp_buffer = NULL;
```

```
883#ifdef LP_STATS
884             lp_table[i].lastcall = 0;
885             lp_table[i].runchars = 0;
886             memset (&lp_table[i].stats, 0, sizeof (struct lp_stats));
887#endif
888             lp_table[i].last_error = 0;
889             init_waitqueue_head (&lp_table[i].waitq);
890             init_waitqueue_head (&lp_table[i].dataq);
891             mutex_init(&lp_table[i].port_mutex);
892             lp_table[i].timeout = 10 * HZ;
893     }
```

Tras esto, registramos el gestor e indicamos qué funciones serán accesibles para el usuario. Para esto, pasaremos una estructura lp_fops a la función de registro en la cual se han indicado estas funciones, tal como explicamos anteriormente.

Si se registró correctamente, pasamos a crear la clase de impresora. En caso de haber un error en la creación de la clase, deshacemos el registro anterior y salimos, y si no lo hay, se sigue adelante.

```
894 /* Registramos el gestor con los puntos de llamada. Se observa que le pasamos un parámetro
lp_fops que es el que indicará qué operaciones podemos hacer con el archivo */
895     if (register_chrdev (LP_MAJOR, "lp", &lp_fops)) {
896             printk (KERN_ERR "lp: unable to get major %d\n", LP_MAJOR);
897             return -EIO;
898     }
899 /* Creamos la clase. Si hubiera algún problema salimos */
900     lp_class = class_create(THIS_MODULE, "printer");
901     if (IS_ERR(lp_class)) {
902             err = PTR_ERR(lp_class);
903             goto out_reg;
904     }
```

El siguiente paso es registrar el driver. Si hubiera algún error, se destruiría la clase y se deshace el registro del gestor de llamadas, caso contrario se sigue adelante.

Si todo ha ido perfectamente, se devolverá 0 o bien error en caso de que no se encuentre el dispositivo o no esté activado el soporte para los diferentes modos de IEEE-1284.

```
/* Registramos el driver. Si no fuera posible, saltará un error y destruiríamos la clase creada antes */
• if (parport_register_driver (&lp_driver)) {
907             printk (KERN_ERR "lp: unable to register with parport\n");
908             err = -EIO;
909             goto out_class;
910     }
912     if (!lp_count) { /* Si no se encontró el dispositivo, saltará un error */
913             printk (KERN_INFO "lp: driver loaded but no devices found\n");
914#ifndef CONFIG_PARPORT_1284
915             if (parport_nr[0] == LP_PARPORT_AUTO)
916                     printk (KERN_INFO "lp: (is IEEE 1284 support enabled?)\n");
917#endif
918     }
920     return 0;
922out_class:
923     class_destroy(lp_class);
924out_reg:
```

```
925         unregister_chrdev(LP_MAJOR, "lp");
926         return err;
927}
```

Bibliografía

Códigos fuente en:

- http://lxr.linux.no/linux+v2.6.25.4/drivers/char/lp.c
- http://lxr.linux.no/linux/include/linux/lp.h

Más información en:

- http://www.leapster.org/linux/kernel/lp/
-

Libro recomendado:

- ***Programación linux 2.0. - API del sistema y funcionamiento del núcleo. Rémy Card, Eric Dumas y Franck Mével.***