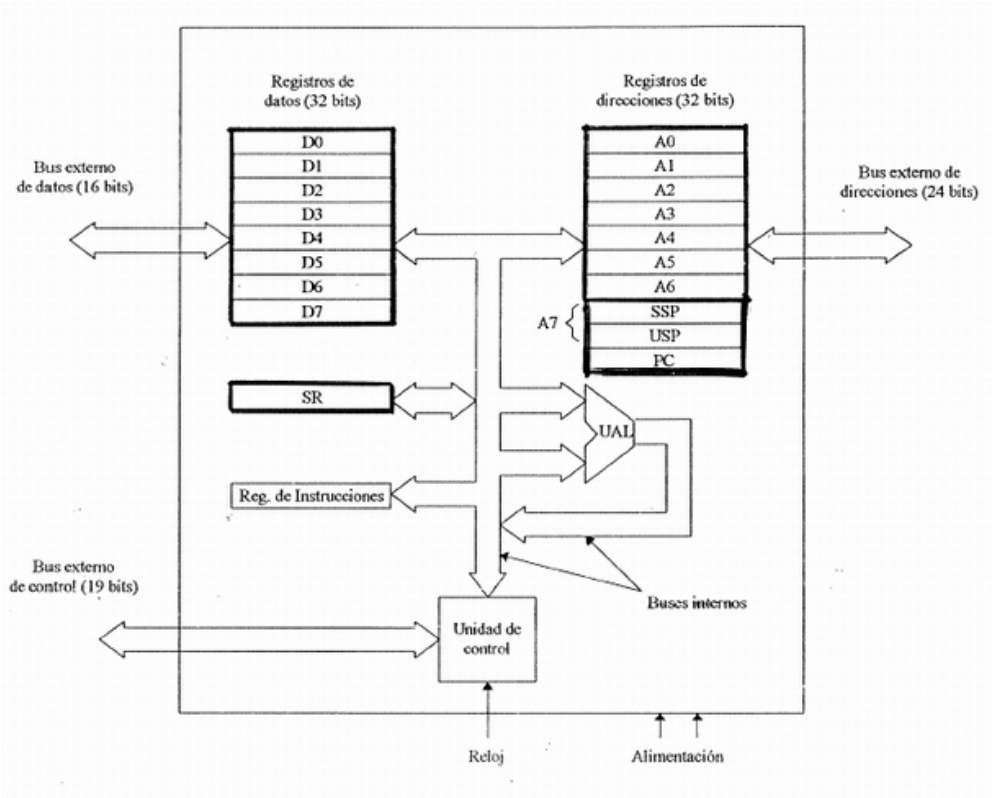
DEVICE DRIVERS EN LINUX

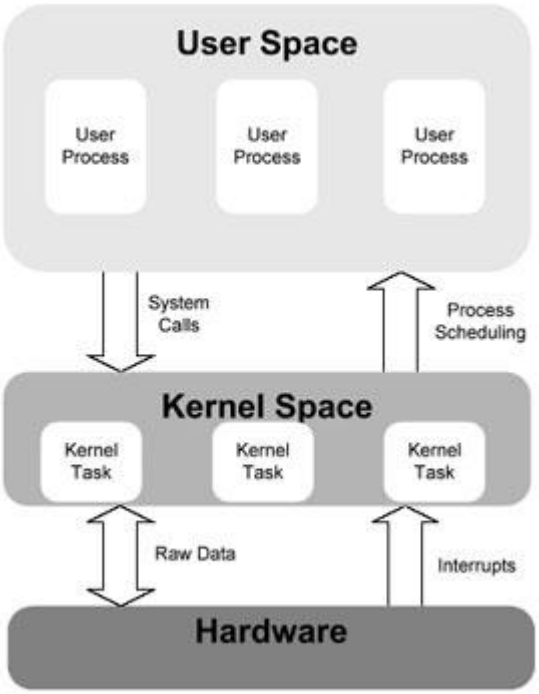
ELEMENTOS ESENCIALES PARA LA CREACIÓN DE DEVICE DRIVERS



DEVICE DRIVERS EN LINUX

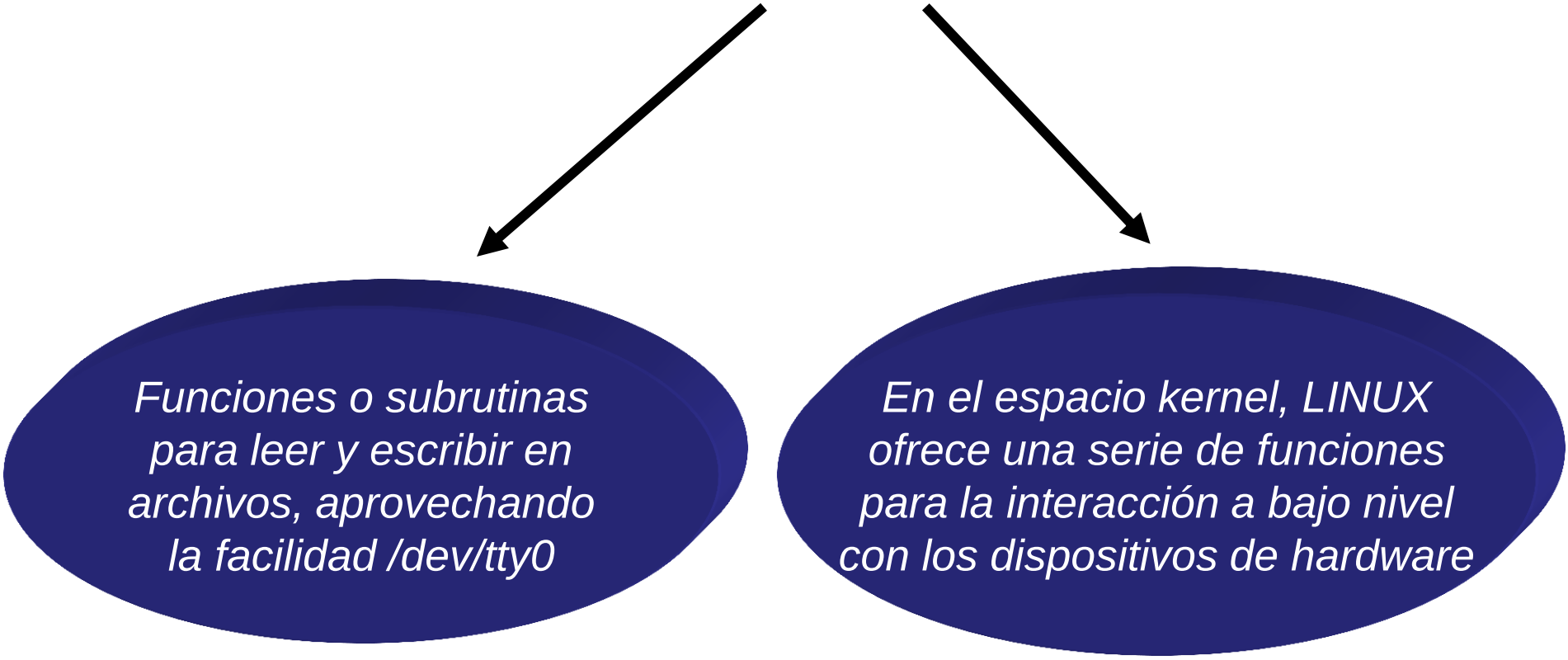
•Espacio del kernel (“kernel space”). El sistema operativo Linux y en especial su kernel se ocupan de gestionar los recursos de hardware de la máquina de una forma eficiente. El kernel, y en especial sus drivers, constituyen así una interfase entre el programador de aplicaciones para el usuario final y el hardware. Toda subrutina que forma parte del kernel tales como los módulos o drivers se consideran que están en el espacio del kernel (“kernel space”).

• Espacio de usuario (“user space”). Los programas que utiliza el usuario final, tales como las “shell” u otras aplicaciones con ventanas como por ejemplo “GEdit”, residen en el espacio de usuario (“user space”). Estas aplicaciones necesitan interaccionar con el hardware del sistema, pero no lo hacen directamente, sino a través de las funciones que soporta el kernel.

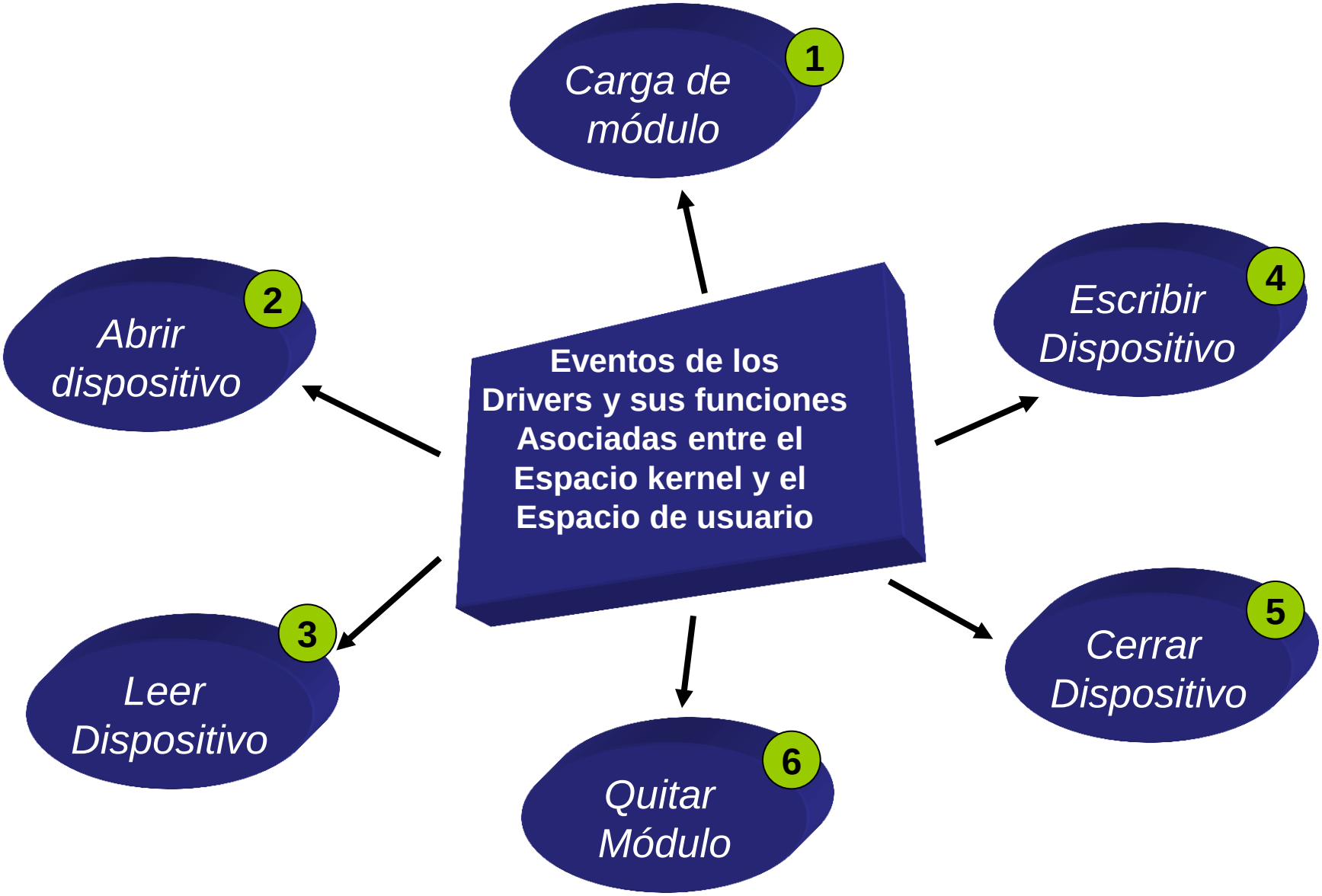


DEVICE DRIVERS EN LINUX

El kernel ofrece una serie de subrutinas o funciones en el espacio de usuario que permiten al programador de aplicaciones finales interaccionar con el hardware.



FUNCIONES DE INTERCAMBIO ENTRE EL ESPACIO DE USUARIO Y EL DEL KERNEL



CARGA Y DESCARGA DEL DRIVER

Cuando un módulo del controlador de dispositivo es cargado dentro del kernel, algunas tareas preliminares se realizan, tales como el reestablecimiento del dispositivo, reservación de interrupciones y reservación de los puertos de entrada y salida, etc.

Tales tareas son realizadas en el espacio Kernel, por dos funciones las cuales necesitan ser representadas (y explícitamente declaradas): init_module y cleanup_module, las cuales se corresponden a los comandos del espacio de usuario insmod y rmmod, los cuales son utilizados cuando se instala o se quita un módulo.

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
  printk("<1> Hello world!\n");
  return 0;
}
static void hello_exit(void)
{
printk("<1>Bye, cruel world\n");
}
init_module(hello_init);
cleanup_module(hello_exit);
}
```

La instrucción printk es similar a la printf a excepción de que trabaja solamente con el kernel. <1> es relativo a la prioridad del mensaje. El mensaje aparece en consola.

UN DEVICE DRIVER SIMPLE

Todo sistema operativo se encuentra equipado con una gran cantidad de rutinas para accesar al hardware, lamados “drivers”. Bajo Linux, éstos son empaquetados en “módulos” o pueden ser compilados de forma directa dentro del kernel. Si se utilizan de manera independiente, éstos pueden ser cargados o descargados en tiempo de ejecución por el kernel mismo (o por cuestiones de uso). Esto hace que el sistema sea más flexible que si el módulo fuese compilado dentro del kernel.

El siguiente es un ejemplo sencillo de la creación de un device driver sencillo:

```
a.c
int init_module()
{
        return 0;
}
#gcc –c a.c
```

Al compilar de esta forma al archivo fuente se tiene el archivo a.o

Se tiene entonces una función muy simple que retorna un valor de tipo entero (cero en este caso), en un archivo llamado ‘a.c’.

Dentro del sistema operativo Linux existe un directorio llamado /proc, el cual contiene gran cantidad de información dentro de la cual está el archivo /proc/modules. Este archivo contiene el nombre (más información adicional) de todos los módulos actualmente cargados en la memoria. Con la instrucción cat, se puede visualizar su contenido.

UN DEVICE DRIVER SIMPLE

```
[joropeza@cluster16 proc]$ more modules
bridge 60381 0 - Live 0xee300000
netloop 11073 0 - Live 0xeelfa000
netbk 79845 0 [permanent], Live 0xee2d8000
blktap 386789 2 [permanent], Live 0xee278000
blkbk 22753 0 [permanent], Live 0xee241000
autofs4 25413 2 - Live 0xeele6000
hidp 24129 2 - Live 0xeelee000
rfcomm 46041 0 - Live 0xee234000
l2cap 31681 10 hidp,rfcomm, Live 0xeel93000
bluetooth 58917 5 hidp,rfcomm,l2cap, Live 0xee224000
sunrpc 158332 1 - Live 0xee250000
iscsi_tcp 27073 0 - Live 0xeelde000
libiscsi 28481 1 iscsi_tcp, Live 0xeel9d000
scsi_transport_iscsi 31305 3 iscsi_tcp,libiscsi, Live 0xeeld5000
scsi_mod 139113 3 iscsi_tcp,libiscsi,scsi_transport_iscsi, Live 0xee201000
ip_conntrack_ftp 12081 0 - Live 0xeel47000
ip_conntrack_netbios_ns 7105 0 - Live 0xeel36000
ipt_REJECT 9665 1 - Live 0xeel4b000
xt_state 6337 7 - Live 0xeel39000
ip_conntrack 56992 3 ip_conntrack_ftp,ip_conntrack_netbios_ns,xt_state, Live 0xeelc6000
nfnetlink 11353 1 ip_conntrack, Live 0xeel43000
iptable_filter 7233 1 - Live 0xeell2000
ip_tables 17669 1 iptable_filter, Live 0xeel3d000
ip6t_REJECT 9537 1 - Live 0xeel32000
xt_tcpudp 7361 24 - Live 0xee0f3000
ip6table_filter 7105 1 - Live 0xee041000
ip6_tables 18821 1 ip6table_filter, Live 0xeel2c000
x_tables 18501 6 ipt_REJECT,xt_state,ip_tables,ip6t_REJECT,xt_tcpudp,ip6_tables, Live 0xeel26000
dm_mirror 33041 0 - Live 0xeel08000
dm_multipath 22601 0 - Live 0xeel01000
dm_mod 61529 2 dm_mirror,dm_multipath, Live 0xeell5000
ipv6 267489 21 ip6t_REJECT, Live 0xeel50000
parport_pc 31205 1 - Live 0xee0f8000
lp 17033 0 - Live 0xee0db000
parport 40841 2 parport_pc,lp, Live 0xee0el000
floppy 58725 1 - Live 0xee0b6000
3c59x 49001 0 - Live 0xee0c7000
mii 9665 1 3c59x, Live 0xee0b2000
ncspkr 7361 0 - Live 0xee052000
```

UN DEVICE DRIVER SIMPLE

Ahora para agregar a.o (el módulo creado anteriormente) dentro de la memoria, se emplean los servicios de un programa especial llamado ‘insmod’ el cual prepara al módulo del archivo (el cual es el archivo objeto), para seccionar ciertas partes del archivo, incrementar parte de su código y cargarlo dentro de la memoria indicándole al kernel en donde lo puede encontrar. Sin embargo, antes de la ejecución de insmod, el código necesita más instrucciones, las cuales son:

char __module_kernel_version[] __attribute__ ((section(".modinfo")))="kernel_version=2.2.12"; int init_module() { return 0; }

Donde se crea la variable __module_kernel_version, la cual se utiliza como un apuntador a un dato de tipo carácter. Al utilizar __atribute__ y section() se crea una sección llamada .modinfo y se inicializa la cadena a “kernel_version=2.2.12”; ahora estamos en la posibilidad de poner el device driver en el archivos modules de la forma:

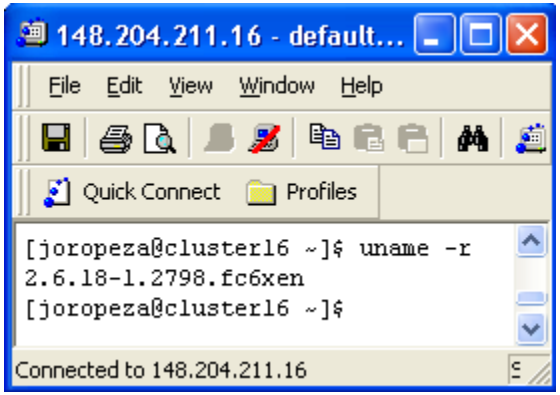
#insmod a

#cat /proc/modules

Para remover el modulo se utiliza la instrucción:

#rnmod a

Si se desea conocer la versión de kernel se utiliza:



UN DEVICE DRIVER SIMPLE

Cuando ‘insmod’ carga el archivo, éste coloca la dirección de la función init_module() en una estructura especial en la memoria. Cuando el kernel desea inicializar el módulo, éste utiliza tal dirección para encontrarla en esta estructura. De tal forma que cuando init_module es llamado por el kernel (y es siempre la primer función a ser llamada), está debe de retornar un 0. Cuando se regresa un 1, el kernel avisa a insmod de tal situación, y éste a su vez despliega el mensaje de error apropiado.

a.c

```
char __module_kernel_version[] __attribute__ ((section(".modinfo")))="kernel_version=2.2.12";
 int init_module()
 {
                printk("Hi\n");
                 return 0;
 }
 #gcc -c a.c #insmod a
```

UN DEVICE DRIVER SIMPLE

```
char __module_kernel_version[] __attribute__ ((section(".modinfo")))="kernel_version=2.2.12";
int init_module()
{
        printk("Hi\n");
        return 0;
}
void cleanup_module()
{
        printk("Bye\n");
}
#gcc -c a.c
#insmod a
#rmmod a
```

La función cleanup_module(), es la última función que se llama antes de que el módulo sea descargado de la memoria.

a.c

```
#include <linux/kernel.h>
#include <linux/module.h>
int init_module()
{
        printk("Hi\n");
        return 0;
}
void cleanup_module()
{
        printk("Bye\n");
}
#gcc -O6 -Wall -DCONFIG_KERNELD -DMODULE -D__KERNEL__ -DLINUX -c a.c
#cat /proc/modules
#insmod a
```

El símbolo <1> indica la prioridad del mensaje, se ha especificado una alta prioridad (bajo número) para que el mensaje aparezca por pantalla y no se quede en los ficheros de mensajes del kernel.

Cuando se cargue y descargue el módulo aparecerán en la consola los mensajes que hemos escrito dentro de printk. Si no los vemos inmediatamente en la consola podemos escribir el comando dmesg en la línea de comandos para verlos o mostrando el fichero de mensajes del sistema con cat /var/log/syslog.

Eventos	Funciones de usuario	Funciones del kernel
Carga del módulo	insmod	init_module
Abrir dispositivo	fopen	Operaciones de archivo: open
Leer dispositivo	fread	Operaciones de archivo: read
Escribir dispositivo	fwrite	Operaciones de archivo: write
Cerrar dispositivo	fclose	Operaciones de archivo: release
Quitar módulo	rmmod	cleanup_module

CREACIÓN DE UN DEVICE DRIVER
DRIVER COMPLETO DE MEMORIA: PARTE INICIAL DEL DRIVER

Se muestra la forma de codificar un device driver: memory.c. Este dispositivo permite que un carácter sea leído o escrito. Este dispositivo, provee un útil ejemplo ya que es un driver completo, a su vez, es fácil de implementar, debido a que no es necesaria una interfaz a un dispositivo de hardware.

PRINCIPIOS DE UN DEVICE DRIVER

```
/* Necessary includes for device drivers */
#include <linux/init.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <asm/system.h> /* cli(), *_flags */
#include <asm/uaccess.h> /* copy_from/to_user */

MODULE_LICENSE("Dual BSD/GPL");

/* Declaration of memory.c functions */
int memory_open(struct inode *inode, struct file *filp);
int memory_release(struct inode *inode, struct file *filp);
ssize_t memory_read(struct file *filp, char *buf, size_t
count, loff_t *f_pos);
ssize_t memory_write(struct file *filp, char *buf, size_t
count, loff_t *f_pos);
void memory_exit(void);
int memory_init(void);
```

```
/* Structure that declares the usual file */
/* access functions */
struct file_operations memory_fops = {
  read: memory_read,
  write: memory_write,
  open: memory_open,
  release: memory_release
};

/* Declaration of the init and exit functions
*/
module_init(memory_init);
module_exit(memory_exit);

/* Global variables of the driver */
/* Major number */
int memory_major = 60;
/* Buffer to store data */
char *memory_buffer;
```

CONEXIÓN DEL DISPOSITIVO CON SUS ARCHIVOS

En LINUX, los dispositivos son accesados de un espacio de usuario de la misma forma como los archivos son accesados. Tales dispositivos de archivo son subdirectorios del directorio /dev.

Para enlazar archivos normales con un módulo del kernel se utilizan dos números: major number y el minor number. El primero es el que utiliza el kernel para enlazar un archivo con su controlador. El segundo número es para uso interno del dispositivo.

To achieve this, a file (which will be used to access the device driver) must be created, by typing the following command as root:

 # mknod /dev/memory c 60 0

In the above, c means that a char device is to be created, 60 is the major number and 0 is the minor number.

CONEXIÓN DEL DISPOSITIVO CON SUS ARCHIVOS

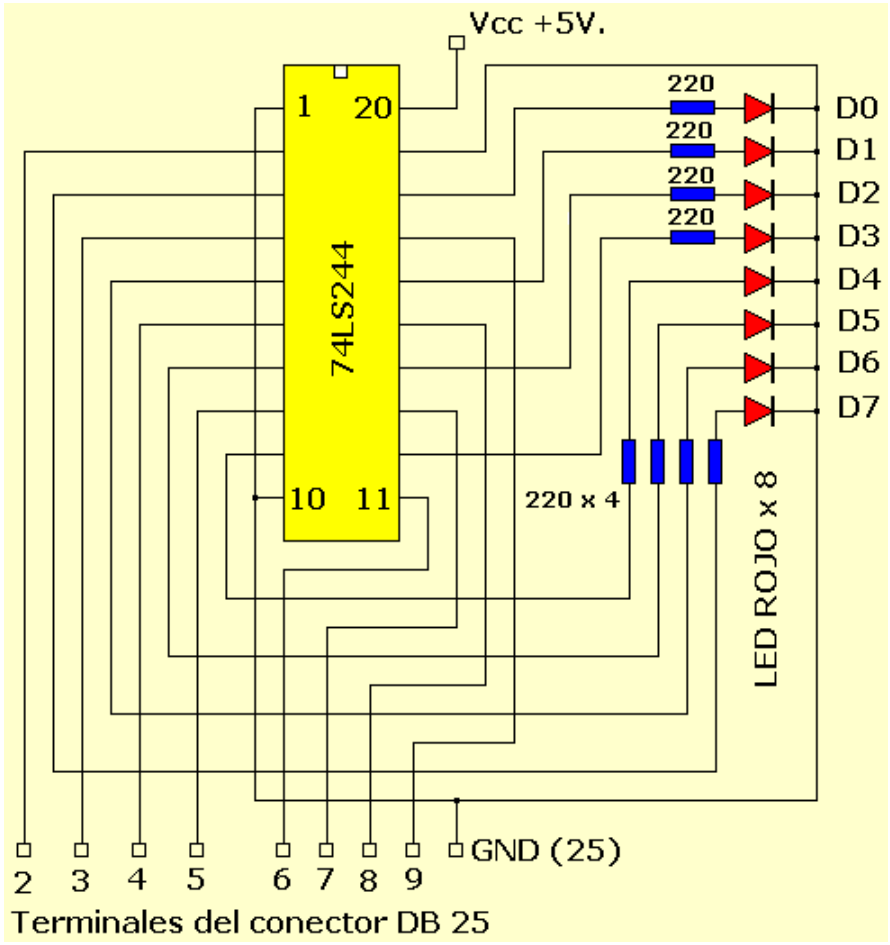
Dentro del controlador, con el objetivo de enlazarlo con su correspondiente archivo /dev en el espacio del kernel, la función register_chrdev es utilizada. Ésta consiste de tres argumentos: major numberm una cadena de caracteres que identifica al nombre del módulo, y la estructura file_operations la cual enlaza la llamada con las funciones de archivo que lo definen. Tal función es invocada, cuando se instala el módulo de la siguiente forma:

```
int memory_init(void) {
  int result;
  /* Registering device */
  result = register_chrdev(memory_major, "memory",
&memory_fops);
  if (result < 0) {
    printk(
      "<1>memory: cannot obtain major number %d\n",
memory_major);
    return result;
  }
  /* Allocating memory for the buffer */
  memory_buffer = kmalloc(1, GFP_KERNEL);
  if (!memory_buffer) {
    result = -ENOMEM;
    goto fail;
  }
  memset(memory_buffer, 0, 1);
  printk("<1>Inserting memory module\n");
  return 0;
  fail:
    memory_exit();
    return result;
}
```

DEVICE DRIVER DEL PUERTO PARALELO

Procederemos ahora a modificar el anterior driver “memoria” para realizar uno que haga una tarea real sobre un dispositivo real. Utilizaremos el ubicuo y sencillo puerto paralelo del ordenador y el módulo se llamará “puertopar”.

El puerto paralelo es en realidad un dispositivo que permite la entrada y salida de información digital. Externamente tiene un conector hembra DB-25 con veinticinco terminales. Internamente, desde el punto de vista de la CPU, ocupa tres bytes de memoria. La dirección base, es decir, la del primer byte del dispositivo, es habitualmente la 0x378 en un PC. En este ejemplo sencillo usaremos únicamente el primer byte. el cual consta enteramente de salidas digitales. La conexión de dicho byte con las terminales del conector exterior aparece en la siguiente figura.



DEVICE DRIVER DEL PUERTO PARALELO

La función init_module anterior del módulo “memoria” habrá que modificarla sustituyendo la reserva de memoria RAM por la reserva de la dirección de memoria del puerto paralelo, es decir, la 0x378. Para ello utilizaremos la función que permite checar la disponibilidad de la región de memoria, check_region, y la función de reserva de una región de memoria para este dispositivo, request_region. Ambas tienen como argumentos la dirección base de la región de memoria y su longitud. La función request_region además admite una cadena de caracteres que define el módulo.

```
<<puertopar modificacion init module>>=
/* Registrando puerto */
port = check_region(0x378, 1);
if (port) {
printk("<1>puertopar: no puedo reservar 0x378\n");
result = port;
goto fallo;
}
request_region(0x378, 1, "puertopar");
```

DRIVER DEL PUERTO PARALELO

```
/* Necessary includes for drivers */
#include <linux/init.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <linux/ioport.h>
#include <asm/system.h> /* cli(), *_flags */
#include <asm/uaccess.h> /* copy_from/to_user */
#include <asm/io.h> /* inb, outb */

MODULE_LICENSE("Dual BSD/GPL");

/* Function declaration of parlelport.c */
int parlelport_open(struct inode *inode, struct file *filp);
int parlelport_release(struct inode *inode, struct file *filp);
ssize_t parlelport_read(struct file *filp, char *buf,
```

DRIVER DEL PUERTO PARALELO

```
                    size_t count, loff_t *f_pos);
ssize_t parlelport_write(struct file *filp, char *buf,
                    size_t count, loff_t *f_pos);
void parlelport_exit(void);
int parlelport_init(void);

/* Structure that declares the common */
/* file access fcuntions */
struct file_operations parlelport_fops = {
 read: parlelport_read,
 write: parlelport_write,
 open: parlelport_open,
 release: parlelport_release
};

/* Driver global variables */
/* Major number */
int parlelport_major = 61;

/* Control variable for memory */
/* reservation of the parallel port*/
int port;

module_init(parlelport_init);
module_exit(parlelport_exit);
```

LA FUNCIÓN init

```
int parlelport_init(void) {
  int result;

  /* Registering device */
  result = register_chrdev(parlelport_major, "parlelport",
      &parlelport_fops);
  if (result < 0) {
    printk(
      "<1>parlelport: cannot obtain major number %d\n",
      parlelport_major);
    return result;
  }

  <parlelport modified init module>

  printk("<1>Inserting parlelport module\n");
  return 0;

  fail:
    parlelport_exit();
    return result;
}
```

ELIMINANDO EL DRIVER

```
void parlelport_exit(void) {

  /* Make major number free! */
  unregister_chrdev(parlelport_major, "parlelport");

  <parlelport modified exit module>

  printk("<1>Removing parlelport module\n");
}
```

ABRIENDO EL PUERTO

```
int parlelport_open(struct inode *inode, struct file *filp) {

  /* Success */
  return 0;

}
```

CERRANDO EL PUERTO

```
int parlelport_release(struct inode *inode, struct file *filp) {

  /* Success */
  return 0;
}
```

LEYENDO DEL DISPOSITIVO

```
ssize_t parlelport_read(struct file *filp, char *buf,
  size_t count, loff_t *f_pos) {

  /* Buffer to read the device */
  char parlelport_buffer;

  <parlelport inport>

  /* We transfer data to user space */
  copy_to_user(buf,&parlelport_buffer,1);

  /* We change the reading position as best suits */
  if (*f_pos == 0) {
    *f_pos+=1;
    return 1;
  } else {
    return 0;
  }
```

ESCRIBIENDO AL DISPOSITIVO

```
ssize_t parlelport_write( struct file *filp, char *buf,
  size_t count, loff_t *f_pos) {

  char *tmp;

  /* Buffer writing to the device */
  char parlelport_buffer;

  tmp=buf+count-1;
  copy_from_user(&parlelport_buffer,tmp,1);

  <parlelport outport>

  return 1;
}
```

Remover los drivers instalados del puerto paralelo

(por ejemplo, lp, parport, parport_pc, etc.).

```
# mknod /dev/parlelport c 61 0

# chmod 666 /dev/parlelport

$ cat /proc/ioports

$ echo -n A >/dev/parlelport
```

APLICACIÓN REALIZADA

```
#include <stdio.h>
#include <unistd.h></p>

int main() {
  unsigned char byte,dummy;
  FILE * PARLELPORT;

  /* Opening the device parlelport */
  PARLELPORT=fopen("/dev/parlelport","w");
  /* We remove the buffer from the file i/o */
  setvbuf(PARLELPORT,&dummy,_IONBF,1);

  /* Initializing the variable to one */
  byte=1;

  /* We make an infinite loop */
  while (1) {
    /* Writing to the parallel port */
    /* to turn on a LED */
    printf("Byte value is %d\n",byte);
    fwrite(&byte,1,1,PARLELPORT);
    sleep(1);

    /* Updating the byte value */
    byte<<=1;
    if (byte == 0) byte = 1;
  }

  fclose(PARLELPORT);

}
```

$ gcc -o lights lights.c