



## UF1287: Desarrollo de componentes software para el manejo de dispositivos (Drivers)

**Certificado de Profesionalidad**  
*IFCT0609 - Programación de sistemas informáticos*



*IFCT0609 > MF0964\_3 > UF1287*

**ic editorial**

M<sup>a</sup> Josefa Díaz Coca

**Desarrollo de  
componentes *software*  
para el manejo de  
dispositivos (*Drivers*)  
IFCT0609**

M<sup>a</sup> Josefa Díaz Coca

**ic editorial**

**Desarrollo de componentes software para el manejo de dispositivos (Drivers).**

**IFCT0609**

**© M<sup>a</sup> Josefa Díaz Coca**

2.<sup>a</sup> Edición

© IC Editorial, 2023

Editado por: IC Editorial  
c/ Cueva de Viera, 2, Local 3  
Centro Negocios CADÍ  
29200 Antequera (Málaga)  
Teléfono: 952 70 60 04  
Fax: 952 84 55 03  
Correo electrónico: [iceditorial@iceditorial.com](mailto:iceditorial@iceditorial.com)  
Internet: [www.iceditorial.com](http://www.iceditorial.com)

**IC Editorial** ha puesto el máximo esfuerzo en ofrecer una información completa y precisa. Sin embargo, no asume ninguna responsabilidad derivada de su uso, ni tampoco la violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Mediante esta publicación se pretende proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para **IC Editorial** ninguna forma de asistencia legal, administrativa ni de ningún otro tipo.

Reservados todos los derechos de publicación en cualquier idioma.

Según el Código Penal vigente ninguna parte de este o cualquier otro libro puede ser reproducida, grabada en alguno de los sistemas de almacenamiento existentes o transmitida por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro, sin autorización previa y por escrito de IC EDITORIAL; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes intencionadamente reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

ISBN: 978-84-1103-950-5

## **Presentación del manual**

El **Certificado de Profesionalidad** es el instrumento de acreditación, en el ámbito de la Administración laboral, de las cualificaciones profesionales del Catálogo Nacional de Cualificaciones Profesionales adquiridas a través de procesos formativos o del proceso de reconocimiento de la experiencia laboral y de vías no formales de formación.

El elemento mínimo acreditable es la **Unidad de Competencia**. La suma de las acreditaciones de las unidades de competencia conforma la acreditación de la competencia general.

Una **Unidad de Competencia** se define como una agrupación de tareas productivas específica que realiza el profesional. Las diferentes unidades de competencia de un certificado de profesionalidad conforman la **Competencia General**, definiendo el conjunto de conocimientos y capacidades que permiten el ejercicio de una actividad profesional determinada.

Cada **Unidad de Competencia** lleva asociado un **Módulo Formativo**, donde se describe la formación necesaria para adquirir esa **Unidad de Competencia**, pudiendo dividirse en **Unidades Formativas**.

El presente manual desarrolla la Unidad Formativa **UF1287: Desarrollo de componentes software para el manejo de dispositivos (Drivers)**,

perteneciente al Módulo Formativo **MF0964\_3: Desarrollo de elementos software para gestión de sistemas**,

asociado a la unidad de competencia **UC0964\_3: Crear elementos software para la gestión del sistema y sus recursos**,

del Certificado de Profesionalidad **Programación de sistemas informáticos**.

# **Índice**

**Portada**

**Título**

**Copyright**

**Presentación del manual**

**Índice**

**Capítulo 1**

**El núcleo del sistema operativo**

- 1. Introducción**
  - 2. Arquitectura general del núcleo**
  - 3. Subsistemas del núcleo**
  - 4. Aspectos de seguridad sobre el desarrollo de elementos del núcleo**
  - 5. Resumen**
- Ejercicios de repaso y autoevaluación**

**Capítulo 2**

**Programación de controladores de dispositivos**

- 1. Introducción**
  - 2. Funcionamiento general de un controlador de dispositivo**
  - 3. Principales tipos de controladores de dispositivos**
  - 4. Técnicas básicas de programación de controladores de dispositivos**
  - 5. Técnica de depuración y prueba**
  - 6. Compilación y carga de controladores de dispositivos**
  - 7. Distribución de controladores de dispositivos**
  - 8. Particularidades en el desarrollo de dispositivos en sistemas operativos de uso común**
  - 9. Herramientas**
  - 10. Documentación de manejadores de dispositivos**
  - 11. Resumen**
- Ejercicios de repaso y autoevaluación**

**Bibliografía**

## Capítulo 2

# Programación de controladores de dispositivos

### Contenido

1. Introducción
2. Funcionamiento general de un controlador de dispositivo
3. Principales tipos de controladores de dispositivos
4. Técnicas básicas de programación de controladores de dispositivos
5. Técnica de depuración y prueba
6. Compilación y carga de controladores de dispositivos
7. Distribución de controladores de dispositivos
8. Particularidades en el desarrollo de dispositivos en sistemas operativos de uso común
9. Herramientas
10. Documentación de manejadores de dispositivos
11. Resumen

## 1. Introducción

Cada dispositivo conectado a un ordenador tiene una forma distinta de relacionarse con el sistema, que dependerá de la naturaleza del mismo.

No se puede tratar de igual manera un dispositivo como, por ejemplo, un ratón, que debe tratar parámetros del tipo: velocidad, posición, etc., que un disco externo, que manejará información relacionada con sectores, pistas, etc.

Por tanto, cada dispositivo necesitará un código específico para controlarlo. Este código es el **controlador del dispositivo o driver**.

Normalmente, estos controladores de dispositivos son implementados por la compañía que los diseña, pero, como los dispositivos se diseñan siguiendo unos estándares *hardware*, que están publicados, pueden ser implementados por otros.

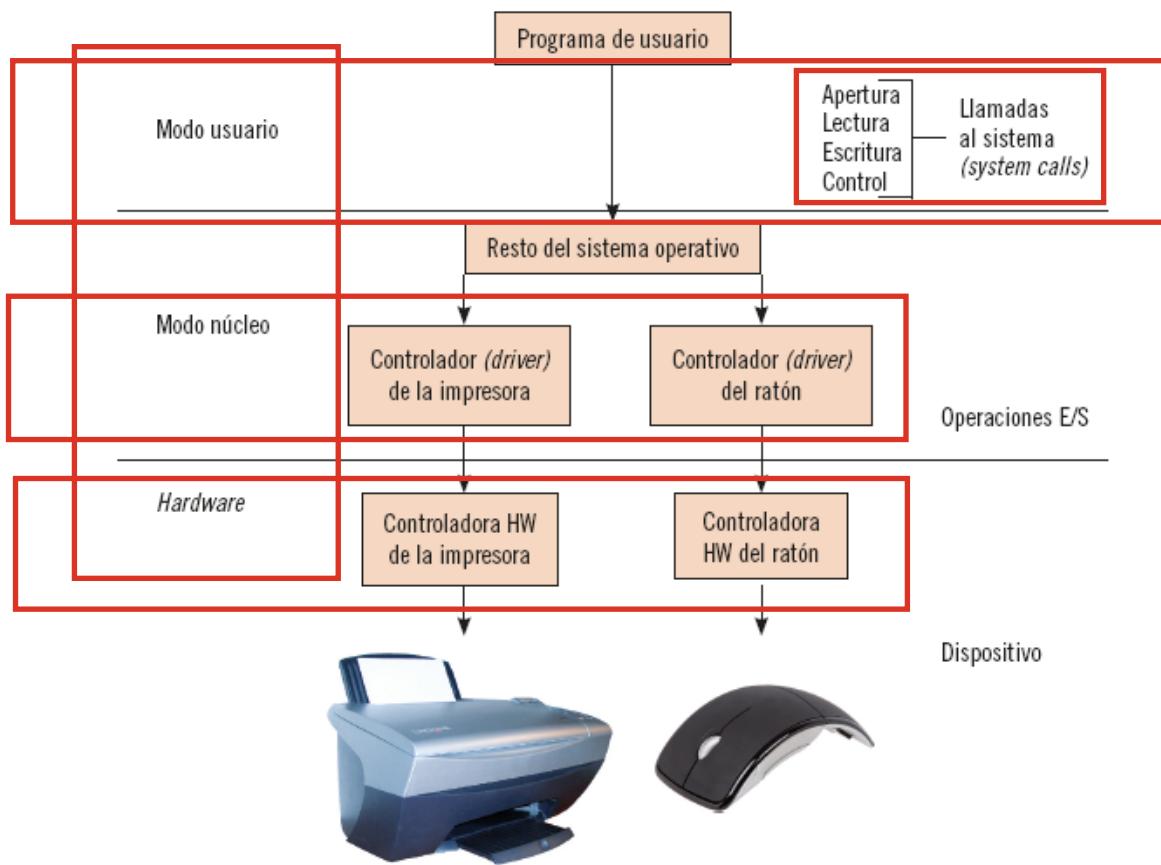
El *driver* necesita acceder al *hardware* del dispositivo y, por esto, normalmente, forma parte del núcleo. También puede formar parte del espacio del usuario, realizando llamadas al sistema para comunicarse con el dispositivo y con esto evitar conflictos, aunque en este capítulo se va a tratar la primera opción, es decir, el *driver* como parte del núcleo.

## 2. Funcionamiento general de un controlador de dispositivo

Un controlador de dispositivos o *driver* es un componente *software* que debe tener acceso a los recursos del sistema. Para esto es necesario que dicho *software* se ejecute en **modo núcleo** o *kernel*.

Por esto, los sistemas operativos están preparados para incorporar los controladores, de forma que puedan interactuar con el dispositivo y con el resto del sistema operativo, tal como se ve en la siguiente imagen.

Esquema general que muestra la posición lógica de los *drivers*



Ejecutan en **modo usuario**, es decir, un entorno restringido, que los aleja de posibles conflictos con el núcleo del sistema operativo.

En el **modo núcleo** o **modo kernel**, donde están incluidos los *drivers*, estos se ejecutan con muy pocas restricciones, lo que implica un gran riesgo, ya que un error aquí puede producir un fallo importante del sistema.

El subsistema de E/S del núcleo se abstrae de gestionar las diferencias entre los distintos dispositivos gracias a los controladores. Estos ocultan estas diferencias proporcionando una interfaz entre el núcleo y los dispositivos, procesando de forma transparente las peticiones de E/S entre ellos.

Además, los controladores de dispositivo proporcionan respuestas a las peticiones que generan los programas de usuario que interactúan con el dispositivo.



### Recuerde

Los controladores de dispositivo proporcionan tres interfaces:

- Con el núcleo del sistema operativo, de manera que comunica las peticiones y permite el acceso a los servicios de este.
- Con el dispositivo, para ejecutar las operaciones solicitadas.
- Con el bus, para gestionar la comunicación con el dispositivo.

Para ampliar información sobre qué es lo que hacen los controladores de dispositivo, se muestra la definición que aparece en *Operating System Concepts* de A. Silberschatz, P. B. Galvin y G. Gagne:

*Un controlador de dispositivo puede ser considerado un traductor. Su entrada consiste en comandos de alto nivel tales como "recuperar el bloque 123". Su salida consiste en instrucciones específicas hardware de bajo nivel que son usadas por la controladora hardware, que conecta el dispositivo E/S con el resto del sistema.*



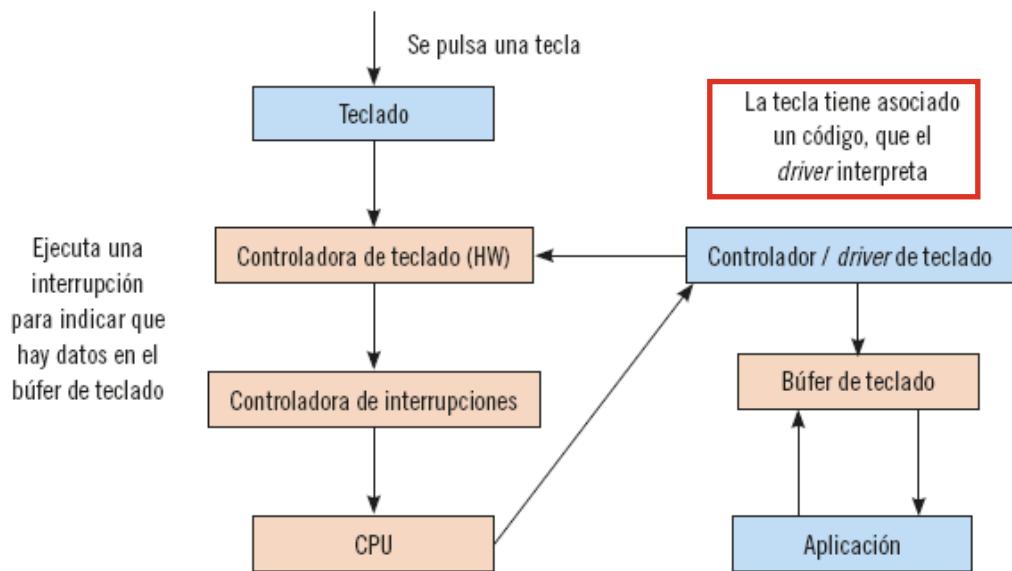
### Actividades

1. Busque una tabla que muestre información sobre las distintas velocidades de los dispositivos más comunes, para observar la gran diferencia entre ellos.
2. Investigue cómo el sistema operativo facilita la instalación de un nuevo dispositivo sin tener que recompilarlo de nuevo.

El funcionamiento, en líneas generales, de los controladores de dispositivo, sin entrar en la parte específica de cada uno de ellos, sigue unas mismas pautas.

Para ver un esquema de este funcionamiento se muestra, a continuación, un gráfico con un ejemplo: el funcionamiento del controlador de un dispositivo de entrada, un teclado.

Esquema general del funcionamiento de un controlador de teclado



### Nota

Se denomina búfer a una **zona de la memoria en la que se almacenan datos de forma temporal** como zona de tránsito, mientras se transmite información entre:

- Dos dispositivos.
- Un dispositivo y una aplicación.

Se utiliza por los siguientes motivos:

- Para adaptar las velocidades entre el dispositivo que escribe en el búfer y el que lee de él.
- Para adaptar el tamaño de transferencia de datos.
- Para garantizar que los datos son los mismos que cuando se realizó la llamada de lectura de estos.

### 3. Principales tipos de controladores de dispositivos

A la hora de clasificar los controladores de dispositivo, se pueden tener en cuenta diversos criterios, por ejemplo:

- La forma en la que **transfieren la información** los dispositivos.
- Si el acceso al dispositivo se hace de forma **secuencial** o de forma **aleatoria**.
- Si las transferencias con el dispositivo se realizan de forma **síncrona** o **asíncrona**.
- Si el dispositivo se **comparte** o está **dedicado**.
- La **velocidad** del dispositivo.
- Según la **forma en la que se accede** al dispositivo:
  - Lectura/escritura.
  - Solo lectura.
  - Solo escritura.



#### Sabía que...

Debido a la gran variedad de tipos, es posible que para un mismo dispositivo existan diferentes controladores.



#### Definición

##### Acceso secuencial

Forma de acceder a un grupo de elementos siguiendo un orden determinado.

##### Acceso aleatorio

Forma de acceder a un elemento de forma arbitraria, en un tiempo constante.

##### Acceso síncrono

Acceso de forma continuada cada determinado período de tiempo fijo.

##### Acceso asíncrono

Forma de acceso sin que dependa de un período de tiempo; se accede bajo una petición.

De forma general, los controladores de dispositivo se pueden categorizar según la forma en la que se produce la transferencia de información entre el dispositivo y el procesador.

A grandes rasgos, se pueden clasificar en los tipos que se describen a continuación.

### 3.1. Carácter

La transferencia de la información se realiza mediante una **secuencia de caracteres**, que debe tener un orden específico.

Ejemplos de este tipo de dispositivos son:

Ratones tambien

- Puertos serie.
- Teclados.
- Impresoras.
- Tarjetas de sonido.

En estos dispositivos, se pueden realizar llamadas básicas al sistema del tipo: lectura y escritura.

Solo tienen una posición, la actual, es decir, no es posible realizar búsquedas.

Además, es posible contar con un búfer para almacenamiento y edición. Un ejemplo de la necesidad de guardar la información en un búfer puede ser cuando se escribe en un teclado y es necesario retroceder y borrar un carácter.

### 3.2. Bloque

Los dispositivos controlados por este tipo de controladores son aquellos capaces de montar un sistema de ficheros.

Ejemplos de este tipo de dispositivos son:

- Discos duros.
- Memorias USB.

La información se transfiere en **bloques de tamaño fijo**, que no necesitan leerse de forma continua; se accede a ellos de un modo aleatorio.

Es posible realizar búsquedas en cualquier posición.

Las llamadas al sistema serán del tipo: **lectura, escritura y búsqueda**.

### 3.3. Paquete

Aunque en este tipo de dispositivos la transferencia de datos se hace también en unidades de tamaño fijo (paquetes), se tratan de forma diferente que en los dispositivos de tipo bloque.



#### Definición

##### Paquetes

Bloques en los que se divide la información que se va a enviar o recibir a través de la red.

Las operaciones de E/S a un dispositivo de bloque, por ejemplo a un disco, son distintas de las que se producen en red, en cuanto a la velocidad y el tipo de direccionamiento. Por esto, se tratan de forma diferente.

Ejemplos de este tipo de dispositivos son:

- Tarjetas Ethernet.
- Tarjetas WiFi.

Una interfaz que está disponible en muchos sistemas operativos son los **sockets** de red.

Un socket es un punto de acceso a una dirección en una aplicación remota. Este acceso se realiza desde un socket local a uno remoto, mediante llamadas al sistema.

Esta conexión permite enviar y recibir paquetes a través de ella y, además, permite permanecer a la espera, hasta que una aplicación remota pueda conectarse.



#### Actividades

3. Busque información sobre un dispositivo que no encaja en la clasificación anterior, los temporizadores o relojes.

## 4. Técnicas básicas de programación de controladores de dispositivos

En este apartado, se van a conocer las técnicas básicas de programación de controladores de dispositivo, para lo que se va a:

- Identificar las **funciones estándar** para manejar los dispositivos independientemente del tipo que sean.
- Analizar la **sincronización** entre el dispositivo y la CPU, mediante el sistema de **interrupciones**.
- Conocer el sistema de **acceso directo a memoria (DMA)** para realizar el paso de información cuando hay un gran volumen de datos.
- Definir la estructura, el funcionamiento y cómo se programan.

### 4.1. Estructuras básicas de datos de dispositivos

Básicamente, entre los dispositivos y el sistema lo que se produce es una transferencia de información, el envío y la recepción de datos.

En esta sección se definen las estructuras básicas de datos que hacen que sea posible esta transferencia.

En primer lugar, se van a definir dos conceptos esenciales, el concepto de dato y el de estructura de datos:

- Se puede definir un **dato elemental** como la unidad mínima de información de la que se dispone en un sistema.
- **Estructura de datos** es la forma en que se organizan un conjunto de datos elementales para que se puedan manipular de forma fácil y eficiente.

Para ver el manejo de los datos de transferencia entre dispositivos, como algo estándar y aislado de las características de cada uno, se van a mostrar una serie de características comunes a todos ellos:

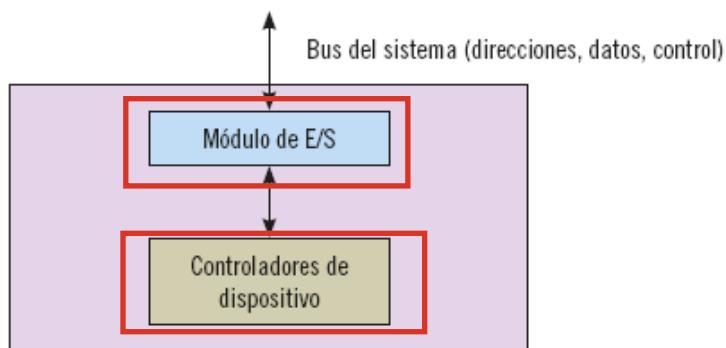
- Debido a las distintas funcionalidades de cada dispositivo, existe una **independencia** entre ellos.
- Existe una diferencia de velocidades de transferencia de datos con el procesador y la memoria. Por esto, es necesario que se produzca una **sincronización de la transmisión**.
- Los datos que transfieren los dispositivos tienen un formato diferente, así como distintas longitudes de palabras; por esto es necesario **adaptar** la forma en que se transmiten estos datos.

- El dispositivo va a ser **identificado** de forma única por parte del sistema, que no va a depender del dispositivo en sí.

Para que sea posible implementar estas características, los sistemas cuentan con dos tipos de elementos que interactúan sobre los dispositivos.

- **Módulo de E/S:** elemento del sistema encargado de gestionar todo lo relacionado con las características comunes de los dispositivos.
- **Controlador de dispositivo:** elemento encargado de gestionar las características específicas.

Esquema de la estructura de E/S



## Módulos de E/S

Los módulos de E/S ofrecen al sistema la posibilidad de gestionar, de forma común, los distintos tipos de dispositivos que se encuentran conectados, dejando los detalles concretos a la parte específica del dispositivo.

La conexión entre el módulo de E/S y el procesador es a través de un bus, que transfiere datos, direcciones y operaciones de control.



### Definición

#### Bus

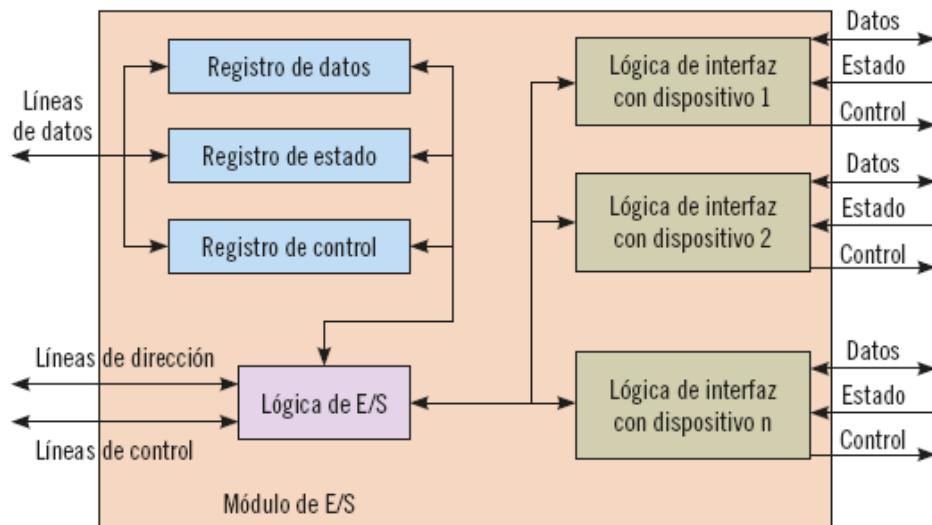
Medio de comunicación entre los diferentes elementos de un ordenador o entre varios ordenadores.

Está constituido por un conjunto de líneas, cables o pistas en un circuito impreso.

Cuando se produce una transferencia de datos, estos se almacenan en un registro de forma temporal, el **registro de datos**. Existen dos registros más, uno para almacenar el estado del módulo y otro que permite que el módulo realice distintas funciones; son el **registro de estado** y el **de control**.

En el esquema que se muestra a continuación, se ve la estructura general de un módulo E/S.

## Esquema de un módulo de E/S



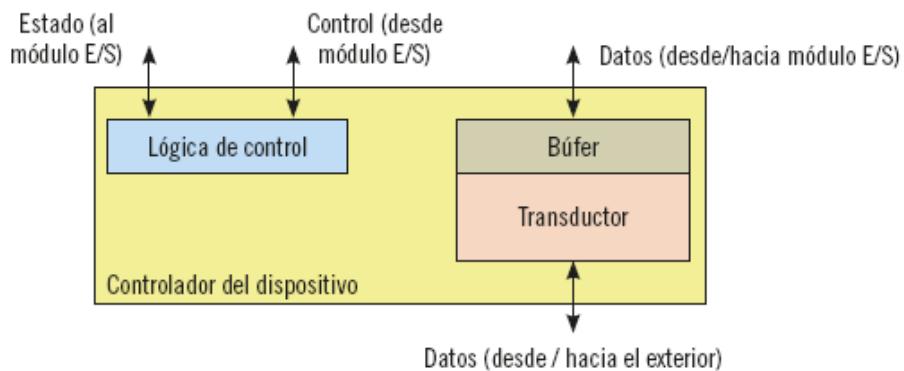
## Controlador de dispositivos

Los controladores de dispositivos tienen como funciones básicas leer datos y realizar peticiones, independientemente del dispositivo del que se trate. Además, debe realizar otras funciones propias del dispositivo, como inicializarlo, etc.

Para comunicarse con el módulo de E/S utiliza las señales de: control, estado y datos.

En el gráfico que se muestra a continuación, se ve una estructura general de un controlador de dispositivo, sin entrar en la parte particular del dispositivo que está gestionando.

### Estructura de un controlador de dispositivo



En el gráfico, aparecen los siguientes elementos:

- **Lógica de control:** está asociada al dispositivo, controla las operaciones y la comunicación con el módulo E/S.
- **Transductor:** convierte señales eléctricas, que están asociadas a los datos, a la forma de energía necesaria para que pasen los datos al módulo de E/S o hacia el exterior.
- **Búfer:** sirve para almacenar temporalmente los datos que el transductor va a convertir.

Hasta ahora se ha visto la parte que está más enfocada al *hardware*. Desde el punto de vista *software*, los controladores de dispositivos tienen entre ellos una estructura parecida.

A continuación, se muestra **un ejemplo** de actuación:

- ✓ • Realizar una comprobación de los **parámetros** de entrada, para ver si son válidos; en caso de que no lo sean, gestiona que se devuelva un **error**.
- ✓ • Comprobar si el dispositivo está o no en uso; si está en uso pasará a una **cola de espera**; si no, se examina el estado del dispositivo para ver si la petición puede procesarse.
- ✓ • Es posible que el dispositivo requiera alguna interacción más, por ejemplo cambiar el estado, etc.
- ✓ • Al terminar, se comprueba que no haya habido errores.



Definición

**Parámetro de entrada**

Información que le llega al código, en este caso al *driver*, y que le sirve para decidir si debe actuar de una manera o de otra.

---

En este ejemplo, se ve un comportamiento muy simplificado, ya que se pueden dar situaciones que provoquen interrupciones y en las que sea necesario parar la ejecución, etc. Todo esto se verá de forma detallada más adelante.

A continuación, se van a detallar las características generales de los controladores de dispositivo, desde el punto de vista del *software*, que hacen que se pueda operar de forma independiente del tipo de dispositivo.

### Interfaz uniforme para los controladores de dispositivos

Para conseguir que existan operaciones comunes, el sistema operativo define una serie de operaciones que cada controlador de dispositivo debe suministrar.

El controlador de dispositivo tiene que hacer la traducción entre la operación estándar y cómo se implementa dicha operación para el dispositivo en cuestión.

Esta traducción se hace con la ayuda de una **tabla de punteros**. Cada puntero apunta a una parte del controlador en la que está el código específico para la operación requerida.

Otra forma de conseguir que los dispositivos se manejen de forma uniforme es a la hora de identificarlos. Para proteger que la forma de identificar los dispositivos sea única, tanto en sistemas *Unix* como en *Windows*, se hace incluyéndolos como objetos identificados en el **sistema de ficheros**.



Definición

#### Tabla

En el ámbito de los lenguajes de programación, se define tabla como un tipo de datos elemental, que corresponde a una colección ordenada de elementos de un mismo tipo.

En C/C++, una variable del tipo tabla que contiene enteros se definiría así:

```
long mi_tabla[3]; /*mi tabla tiene 3 elementos*/  
/*para acceder a los datos de la variable mi_tabla*/  
mi_tabla[0] = 1;  
mi_tabla[1] = 2;  
mi_tabla[0] = 3;
```

## Puntero

Tipo de datos elemental. Una variable del tipo puntero contiene la dirección de otra variable.

Si P es un tipo de datos cualquiera, P\* es un tipo puntero que apunta a P.

En C/C++, sería así:

```
long mi_numero = 5;  
  
long * p; /* p es un puntero a long*/  
  
p = &mi_numero /*p contiene la dirección de mi_numero*/
```

## Tabla de punteros

Variable tipo tabla, en la que sus elementos son punteros.

Un ejemplo en C/C++ se declararía así:

```
long*[3] mi_tabla2; /* tabla de 3 elementos tipo  
punteros */
```

## Buffering

Es una técnica que se utiliza para evitar bloqueos o reducir las diferencias de velocidades. Se utiliza una zona de memoria donde se guardan los datos pendientes de transferencia entre el dispositivo y el sistema o viceversa, esta zona es el **búfer**. Las peticiones se ejecutan en el orden en el que llegaron.

El uso de esta técnica implica la copia de datos que puede hacer que se resienta el rendimiento del sistema.

Un ejemplo de dispositivos que usan esta técnica son los dispositivos de sonido. **Y de video**

## Spooling

pej las colas de impresión

Esta técnica consiste en **utilizar el disco fijo como búfer**, de manera que no hay problemas de almacenamiento ni de orden de acceso.

El dispositivo accede a los datos cuando esté preparado para hacerlo.

Un ejemplo de dispositivos que trabajan con esta técnica son los de impresión.

Con esta técnica, es posible que varios dispositivos puedan estar enviando información al búfer sin que se mezcle.

Cada petición se procesará a su velocidad sin interferir en las demás peticiones.

## Independencia del tamaño del bloque del dispositivo

Esta es una técnica que permite que, en la **capa** que está por encima de los controladores de dispositivos, los bloques **son los que se trabaje sean abstractos o lógicos**, de forma que **siempre tengan el mismo tamaño**. Por ejemplo, si el sistema pretende leer de dos dispositivos de disco distintos, es necesario tener en cuenta que ambos tienen un tamaño de **sectores** diferente.



### Nota

En los dispositivos de almacenamiento, por ejemplo en los discos magnéticos, los datos se transfieren de forma lógica en bloques y estos se almacenan en unas zonas físicas, llamadas sectores, con el mismo tamaño de estos bloques.

Para que la transferencia de datos pueda ser tratada de la misma manera, cada uno de los discos va a llenar el bloque lógico con la cantidad de datos necesarios para completar el tamaño fijo de este bloque.

Lo mismo pasará si estamos tratando un dispositivo de tipo carácter. En este caso, se adaptará el tamaño en que se parten las cadenas de caracteres para gestionarlas, a un tamaño fijo y lógico, con el que va a trabajar la capa superior.



### Importante

Investigue sobre cómo está constituido un dispositivo muy importante y que forma parte de la jerarquía de memoria de un computador: el disco magnético.



### Actividades

4. Investigue sobre cómo está constituido un dispositivo muy importante y que forma parte de la jerarquía de memoria de un computador: el disco magnético.

## 4.2. Gestión de errores de dispositivos

Hay situaciones en la utilización de operaciones de E/S, para manejar dispositivos, que hacen que los errores sean muy frecuentes. Por ejemplo: el acceso a los distintos dispositivos, las posibles interferencias entre procesos que quieren acceder al mismo, etc.

Aunque haya mecanismos para intentar evitar los conflictos, desde el *software* y desde el *hardware*, es común encontrarse con errores.

Por esto, es muy importante controlar todas las posibles situaciones de conflicto que puedan desencadenar un error.

Si se producen errores, es necesario identificarlos e informar de este suceso, para poder tratarlo en consecuencia.

Los errores específicos del dispositivo serán tratados por el controlador de este, pero hay otros tipos de errores cuya gestión es independiente del mismo.

Se pueden encontrar, a grandes rasgos, los siguientes tipos de errores:

- **Errores de programación:** ocurren cuando, dentro del código, aparecen bucles que dependen de una condición imposible de satisfacer. Por ejemplo: intentar escribir en un dispositivo de entrada, como un ratón, un teclado, etc.

• **Errores en los parámetros:** pueden aparecer si el parámetro de entrada o el parámetro que se obtiene como salida incurren en un error. Por ejemplo: devolver una dirección de memoria a la que no se puede acceder. En ambos casos, la forma de gestionar el error es enviar un mensaje que indique que se ha producido un error y devolver el código del mismo.

• **Errores propios de las E/S:** aparecen a consecuencia del manejo específico del dispositivo. Por ejemplo: intentar leer de una cámara que está apagada. En este caso, el controlador del dispositivo, en el ejemplo el de la cámara, sabrá cómo actuar.

• **Errores que no pueden ser clasificados** en las categorías anteriores: aquellos que afectan a las estructuras de datos, errores en el manejo del sistema de directorios, etc. En este caso, el sistema se encarga de gestionarlos según su gravedad, enviando un mensaje o pantalla de error, procediendo a cerrar el proceso implicado en la transferencia con el dispositivo o incluso el propio sistema.

### Aplicación práctica

Se dispone de una lista de errores producidos y se necesita clasificar por el tipo de error para tratar de elaborar el plan de actuación correspondiente. La lista de errores es la siguiente:

- Enviar un texto para visualizar a un dispositivo de tipo teclado. error de programacion
- Intentar leer de una llave USB que no está conectada. Errores propios de E/S
- Imprimir un documento y la impresora está apagada. Errores propios de E/S
- Se pretende utilizar de forma errónea la estructura de directorios. Errores pueden ser clasificados
- Se ha intentado escribir en una dirección de memoria protegida. Error de programacion
- En una función que tiene que devolver un mensaje de éxito o fracaso, se devuelve una dirección de memoria. Error de parametros

### Solución

Para mostrar la clasificación de los errores enviados en la lista anterior, se va a utilizar una tabla:

Error	Clasificación
Enviar un texto para visualizar a un dispositivo de tipo teclado.	Errores de programación
Intentar leer de una llave USB que no está conectada.	Errores propios de la E/S
Imprimir un documento y la impresora está apagada.	Errores propios de la E/S
Se pretende utilizar de forma errónea la estructura de directorios.	Errores sin clasificar a resolver por el sistema
Se ha intentado escribir en una dirección de memoria protegida.	Errores de programación
En una función que tiene que devolver un mensaje de éxito o fracaso enviar una dirección de memoria.	Errores en los parámetros

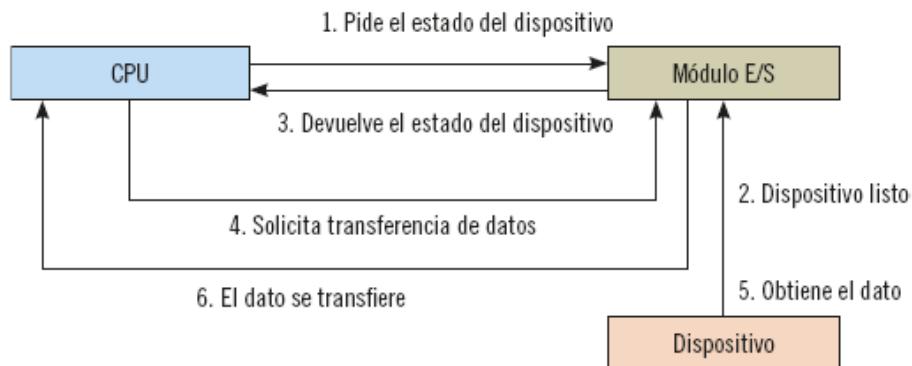
### 4.3. Gestión de memoria de dispositivos

La memoria es uno de los recursos internos esenciales del sistema, que debe ser compartido.

Las diferentes velocidades entre la memoria y los dispositivos hacen que la transferencia de datos entre ellos sea muy importante de cara a optimizar el rendimiento general del sistema.

En la imagen que aparece a continuación, se muestra un ejemplo de cómo transcurrirían las transferencias de datos y las peticiones entre el procesador y los dispositivos.

Esquema de ejemplo de secuencia de transferencia de datos entre la CPU y un dispositivo



Para mejorar el rendimiento y gestionar este paso de datos entre los dispositivos y la memoria, se utilizan varias soluciones:

tres niveles de memoria caché

- Zonas intermedias de paso y almacenamiento de los datos, como son caché y otras formas de almacenamiento intermedio.
- Buses de interconexión que ofrecen una mayor velocidad, mediante el uso de estructuras más sofisticadas.
- Multiprocesadores que mejoren la gestión de la gran cantidad de demandas de E/S.



## Definición

### Caché

Búfer especial que funciona de forma similar a la memoria principal, pero es de menor tamaño y su acceso es más rápido.

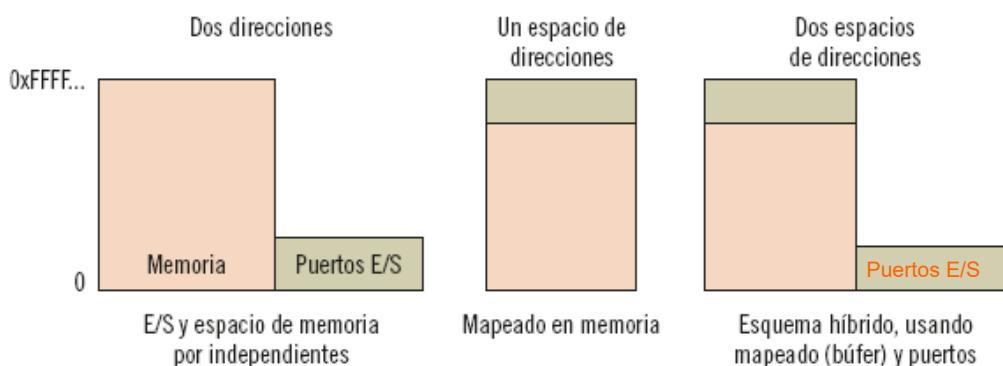
Como se ha visto anteriormente, la transferencia de datos entre el dispositivo y la CPU no puede realizarse directamente, por diversas razones, por ejemplo debido a la velocidad del dispositivo o a la falta de previsión a la hora de realizar una respuesta por parte de este.

Esto hace que sea necesario un mecanismo que sincronice este paso de datos, para lo que se utiliza, como se ha visto, la interfaz de E/S. Cada interfaz utiliza una serie de registros y un búfer para comunicarse con la CPU.

Existen dos formas de comunicación entre la CPU y la interfaz, que depende de la arquitectura del procesador:

- **Conexión mapeada en memoria:** el circuito interfaz utiliza unas direcciones especiales, conectándose como si se tratase de una dirección de memoria. A cada registro de control se le asigna una dirección de memoria única, un puerto mapeado en memoria. Las direcciones asignadas normalmente se encuentran en la zona superior de la memoria.
- **Conexión mediante puertos de E/S:** el circuito interfaz se conecta a través de líneas especiales a un puerto de E/S. Es posible conectarse a ellos mediante unas instrucciones especiales. Para establecer la comunicación, los registros de control tienen asignado un número de puerto. El conjunto de todos los números de puerto asignados a los registros del dispositivo forma el **espacio de puertos de E/S**. A este espacio solo puede acceder el sistema operativo, ya que está protegido de los programas de usuario.

Esquema que muestra distintos tipos de conexiones por mapeo, por puertos E/S y uno híbrido



### Nota

Un puerto de E/S es una dirección que permite a los controladores de dispositivos comunicarse con el dispositivo *hardware*.

El funcionamiento en ambos casos es el siguiente:

- La CPU quiere leer un dato y pone la dirección de la que necesita leer, ya sea de un puerto o de memoria, en el bus de direcciones.
- Lanza una señal que indica la operación que quiere realizar, que en este caso es de lectura, en el bus de control (READ).
- Se envía una señal que indica si se trata de un puerto o una dirección de memoria.
- Si se trata de una dirección de memoria, esta responde a la petición; si se trata de un puerto E/S, es el dispositivo quien responde a la petición.

Existen algunas ventajas a la hora de utilizar la conexión mapeada en memoria, ya que no es necesario utilizar código ensamblador. En este caso, el acceso a las direcciones se haría de la misma forma en que se accede a cualquier variable, de manera que para lectura y escritura se pueden utilizar instrucciones en C o C++.

Además, no sería necesario programar un mecanismo especial para proteger estas direcciones de memoria, de las que utilicen otros procesos de usuario; sería el sistema operativo quien se encargaría de esta protección.

También tiene sus desventajas, ya que en sistemas con caché, al leer de ahí los datos, en lugar del dispositivo, si el dispositivo cambia, la información que se ha leído no es correcta.

Para evitar esto, debería ser posible deshabilitar la caché mientras se estén ejecutando determinadas acciones, por ejemplo si los datos influyen en la condición de salida de un bucle.

De una forma práctica, es posible ver los puertos asignados a un dispositivo:

#### Para usuarios de Windows

Es posible ver los puertos asignados a un dispositivo mediante el comando **msinfo32.exe**.

Para utilizarlo, hay que acceder a la consola del sistema:

- **Inicio → cmd** y se pulsa la tecla [Enter] ↴



Consola de Windows en la que se ha ejecutado el comando msinfo32

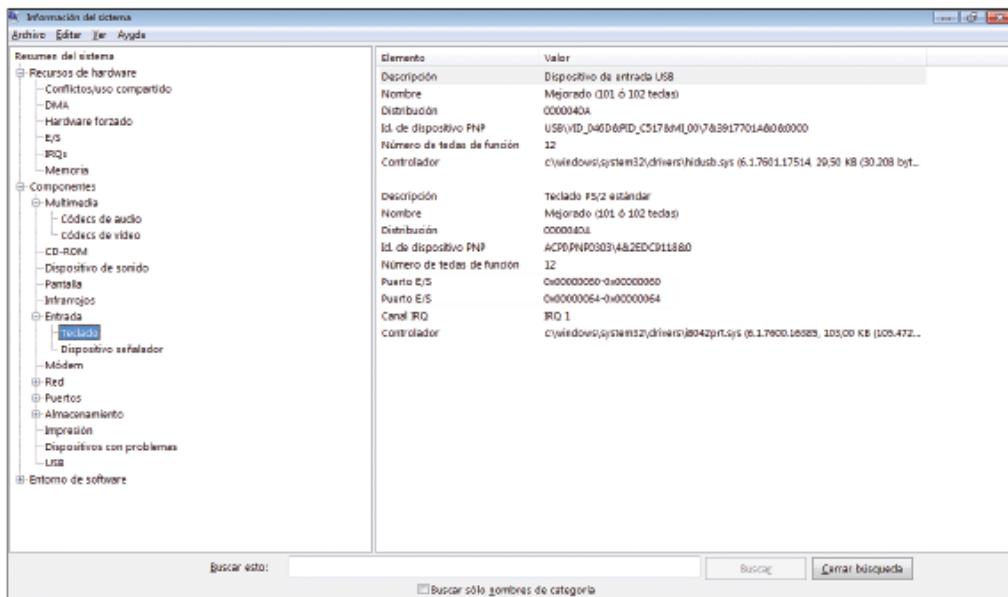
Una vez ejecutado este comando, aparece una pantalla que muestra información de todo el sistema.

En esta pantalla, se selecciona, por ejemplo, un dispositivo de entrada: el teclado.

Se pulsa sobre él y aparece toda la información disponible para este dispositivo.

Entre esta información pueden verse las direcciones de los puertos E/S que tiene asignado.

Esta información aparece en la imagen que se muestra a continuación.

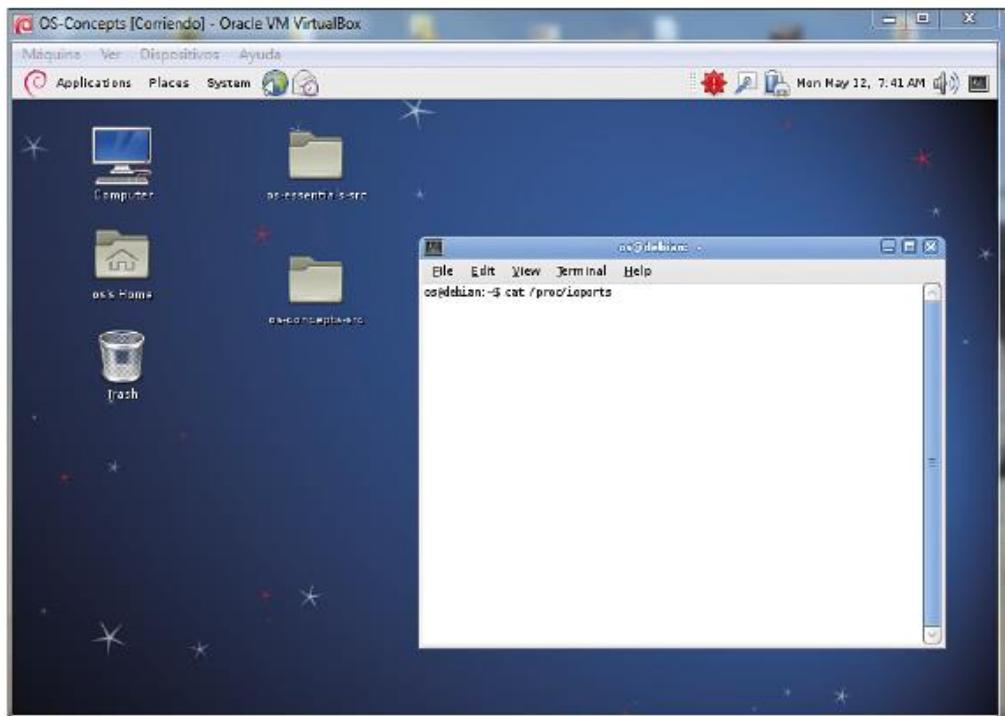


Pantalla de información del sistema en Windows 7

### Para los usuarios de Linux

Se pueden ver los puertos asignados a los dispositivos consultando el fichero "/proc/iports".

Para esto se utiliza, desde un terminal en *Linux*, el comando: **cat /proc/iports** y se pulsa la tecla [Enter] .-.



Terminal del sistema operativo *Debian* (*Linux*)

En la siguiente imagen se muestra la pantalla del terminal del sistema *Linux* (en este caso *Debian*), en la que aparece la asignación de los puertos de los dispositivos.

Enmarcada aparece la dirección asignada a los puertos del dispositivo de entrada: teclado.

```
Máquina Ver Dispositivos Ayuda
Applications Places System Mon May 12, 7:42 AM
File Edit View Terminal Help
0064-0064 : keyboard
0070-0071 : rtc_cmos
0070-0071 : rtc0
0080-008F : dma_page_req
00a0-00a1 : pic2
00c0-00dF : dma2
00f0-00FF : fpu
0170-0177 : 0000:00:01.0
0170-0177 : ata_pii
01f0-01f7 : 0000:00:01.1
01f0-01f7 : ata_pii
0375-0375 : 0000:00:01.0
0376-0376 : ata_pii
03c0-03df : vga+
03f5-03ff : 0000:00:01.0
03f6-03f6 : ata_pii
0cf0-0cff : PCI config
4009-4009 : ACPI PM1a_EVT_BLK
4004-4005 : ACPI PM1a_CNT_BLK
4008-400b : ACPI PM_TMR
4028-4021 : MCP SPEO_BLK
d009-d00f : 0000:00:01.0
d009-d00f : ata_pii
d010-d017 : 0000:00:03.0
d010-d017 : e1000
d029-d03f : 0000:00:04.0
d109-d1ff : 0000:00:05.0
d100-d1ff : Intel 82801AA-ICH
d209-d23f : 0000:00:05.0
d249-d247 : 0000:00:0d.0
d240-d247 : ahci
d250-d257 : 0000:00:0d.0
d250-d257 : ahci
d260-d26f : 0000:00:0d.0
d260-d26f : ahci
root@debian:~$
```

Terminal del sistema operativo Debian (Linux) ejecutando la función: `cat /proc/ioports`

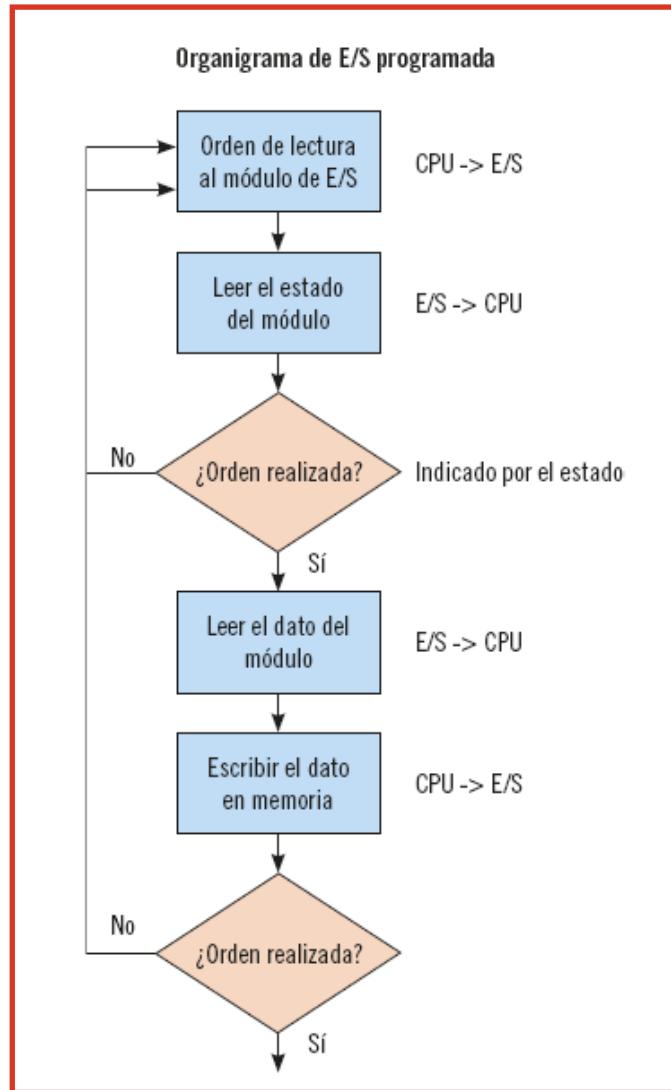
Para finalizar el apartado, cabe señalar que un elemento importante para la gestión de la memoria es el DMA (*Direct Memory Access*), que se verá más adelante.

#### 4.4. Control de interrupciones

Para que las operaciones de E/S enviadas desde los dispositivos influyan en la CPU, existen dos mecanismos básicos:

##### Sincronización por programa (E/S programada)

Este tipo de sincronización es fácil de implementar. Su comportamiento puede verse en el siguiente organigrama.



Si el dispositivo no está disponible, permanece a la espera hasta que el estado del módulo sea el correcto. Si todo es correcto, el ciclo continúa con la lectura del dato desde el dispositivo de E/S hacia la CPU y, posteriormente, si es posible, con la escritura del dato en memoria desde la CPU (si no, permanece de nuevo a la espera).

Este mecanismo tiene varios inconvenientes:

- Mientras se encuentra en espera, la CPU no puede realizar ninguna operación más y, por lo tanto, se produce una pérdida de tiempo.
- No es posible que se realicen tareas periódicas, ya que hasta que no se libere la CPU no se da paso a la siguiente operación de E/S. Por ejemplo: la operación de refresco de una pantalla.
- Cuando hay varias peticiones de distintos dispositivos, no pueden realizarse a la vez, ya que hasta que no termine uno no empieza el siguiente dispositivo.

**Estos dos últimos inconvenientes pueden verse mejorados gracias a la limitación de los tiempos de espera (timeout),** de manera que si el dispositivo queda a la espera más de un período determinado de tiempo se pasa a la siguiente operación de E/S.



## Actividades

- Amplíe información sobre las E/S programadas, buscando información sobre las E/S en serie y en paralelo.

Ratón:

$$400 \text{ (unidades)} * 30 \text{ (veces x segundo)} = 12000 \text{ ciclos} / 1.000.000 = 0.012 \text{ mhz}$$

$$500 - 100\% \\ 0,012 - x\% -- 0,0024\%$$

## Aplicación práctica

Determine el porcentaje de tiempo que está ocupando la CPU con los siguientes dispositivos: un ratón y un disco duro.

Se supone que, para conectarse a la CPU, los dispositivos lo hacen mediante una E/S programada.

El ciclo completo de la operación de E/S es de 400 unidades y la CPU trabaja a 500 MHz.

Hay que tener en cuenta lo siguiente:

- Es necesario leer el ratón 30 veces por segundo para que no se pierda ningún movimiento.
- El disco duro hace la transferencia de datos con el procesador en bloques de 4 palabras y transfiere a una velocidad de 4 Mb/s. La información no debe perderse.

### Solución

#### Ratón

Para que se pueda leer sin perder ningún movimiento, es necesario realizar la operación de lectura 30 veces por segundo. Como el ciclo completo de la operación es 400, se va a calcular cuántos ciclos por segundo se necesitarán:

$$\text{Nº ciclos de reloj por segundo para la lectura} = 30 \times 400 = 12.000 \text{ ciclos por segundo}$$

El porcentaje de ciclos de procesador consumidos es:

Teniendo en cuenta que:

$$12.000 \text{ ciclos por segundo} = 0,012 \text{ MHz}$$

(Nota: La conversión de 1 MHz a ciclo por segundo es:  $1 \text{ MHz} = 10^6 \text{ ciclos por segundo}$ ).

Si el tiempo total del procesador es 500 MHz, el % de este tiempo que utiliza el ratón es:

$$\begin{aligned} 500 \text{ MHz} &\longrightarrow 100 \% \\ 0,012 \text{ MHz} &\longrightarrow x \\ \frac{0,012 \cdot 100}{500} &= 0,0024 \% \end{aligned}$$

El porcentaje de tiempo de CPU utilizado por el ratón es muy bajo: 0,0024 %.

#### Disco duro

La lectura del dispositivo tiene que hacerse para que pueda leer los 4 bloques (palabras). Se sabe que la velocidad de transferencia es de 4 Mb/s.

Estos 4 Mb se dividen por los 16 bytes en que se realiza la transferencia:

$$\begin{aligned} \frac{4 \text{ Mb/s}}{16 \text{ byte}} &= 0,25 \text{ Mb}, \text{ se convierte esto en Kbytes} \\ 0,25 \cdot 1000 &= 250 \text{ K} \end{aligned}$$

Luego la lectura se realiza a 250 K veces por segundo.

Como el tiempo que se invierte en una operación de E/S es de 400 ciclos, se calculan los ciclos que se consumen para la lectura =  $250 \text{ K} * 400 \text{ ciclos/s} = 250 * 400 = 10^6 \text{ ciclos por segundo}$ .

$$\begin{array}{l} 500 \text{ MHz} (500 \cdot 10^6 \text{ cps}) \longrightarrow 100 \% \\ 10^6 \text{ cps} \longrightarrow x \end{array}$$

$$x = \frac{(10^6 \cdot 100)}{500 \cdot 10^6} = 20 \%$$

El porcentaje de tiempo de CPU utilizado por el disco duro es: 20 %, de manera que el procesador tiene que invertir una quinta parte del tiempo en este dispositivo. Por lo tanto, para este caso, no es eficiente este tipo de E/S programada.

### Sincronización por interrupción

Para aprovechar el tiempo de espera que se desperdicia en las E/S programadas se incluyó el sistema de interrupción.

Cuando se está ejecutando un programa y se produce un suceso, este genera una señal externa, que provoca que se interrumpa la ejecución del código de forma temporal. Además, se produce una bifurcación a una dirección específica de memoria.

A partir de esta dirección de memoria, se encuentra la **rutina** que se encarga de manejar el suceso, lo que se denomina **interrupción**, es decir, esta rutina realizará la operación de E/S y, una vez que ha terminado, devuelve el control al programa interrumpido en el punto en el que se quedó.

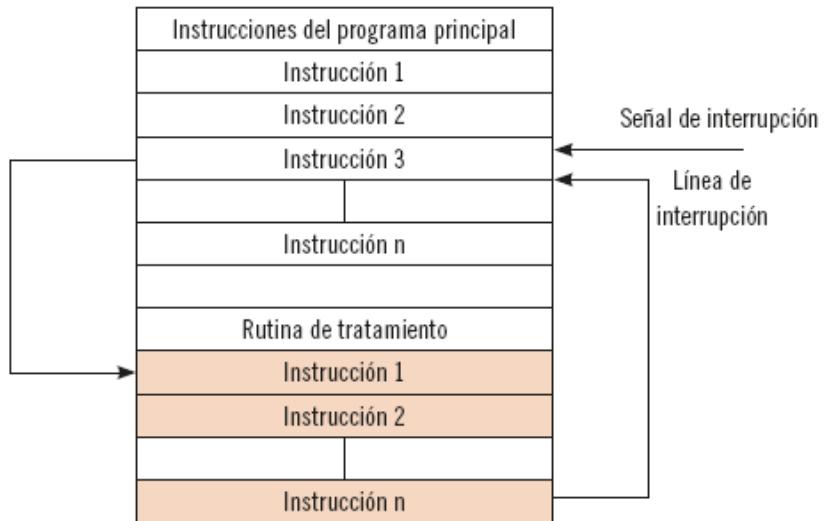


#### Definición

##### Rutina

Conjunto de instrucciones, dentro de un mismo programa, que pueden llamarse para ser ejecutadas, desde las zonas del programa que sean en las que se requiera.

**Línea de ejecución de un programa que es interrumpido  
y ejecuta la rutina de tratamiento de dicha interrupción**



De esta forma, la CPU solo deja de procesar las instrucciones cuando se produce una interrupción, pasando así a ejecutar la subrutina que tiene asociada, continuando la ejecución por el sitio donde lo dejó.

Esto hace que se eliminen los tiempos de espera.

Para que este mecanismo pueda ejecutarse, es necesario que exista un sistema de interrupciones *hardware* capaz de hacer que el programa pueda dar el salto a la rutina especificada cuando sea necesario.

Desde el punto de vista del *hardware*, las interrupciones funcionan de la siguiente manera. Cuando un dispositivo E/S termina una petición, se provoca una interrupción y ocurre lo siguiente:

- Se envía una señal que es detectada por el controlador de interrupciones y este decide qué hacer.
- Si no hay más interrupciones pendientes, se procesa la que acaba de suceder de forma inmediata; si no, dependerá del nivel de prioridad que tenga ese dispositivo.
- El controlador pone un número en el bus de direcciones que indica cuál es el dispositivo que ha terminado la operación.
- Envía una señal para interrumpir la CPU.
- La CPU se para (salvando automáticamente el contador del programa y el registro de estado).

- El número del bus de direcciones es un índice a una tabla que contiene **vectores de interrupción** y que indica el nuevo contador de programa, que va a la rutina de la interrupción.
- Se ejecuta la rutina de la interrupción y se avisa de que ya se está disponible para otra interrupción.
- La CPU vuelve al programa interrumpido.



### Aplicación práctica

**Se está utilizando un sistema que no permite el tratamiento hardware de interrupciones. La idea es simular el comportamiento que realizaría el sistema de interrupciones sin tenerlo implementado.**

**¿Se podría conseguir un efecto similar a la gestión de interrupciones en un procesador sin este sistema? Si es así, ¿cómo?**

### SOLUCIÓN

En un sistema de interrupciones es necesario introducir al final de la ejecución de cada instrucción una consulta a las líneas de interrupción.

En un sistema sin interrupciones, se puede simular este comportamiento, añadiendo, detrás de cada instrucción del programa, una consulta, para ver si se ha producido una operación E/S y, en caso afirmativo, dirigir el flujo del programa hacia la rutina que maneje la operación. De esta manera, la CPU iría ejecutando instrucciones del programa cuando el dispositivo no estuviera disponible. Pero esta implementación sería poco eficiente.



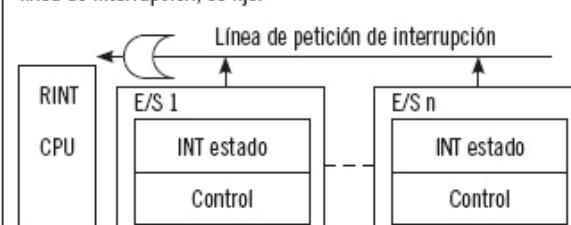
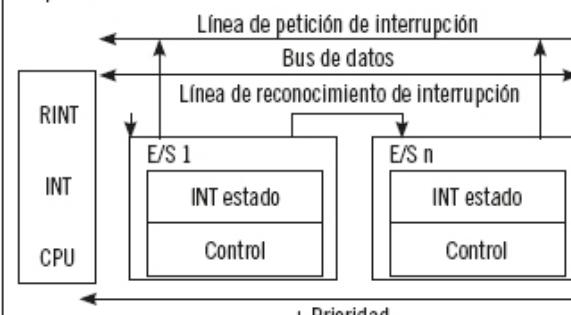
### Actividades

6. Busque una tabla que muestre las fuentes de interrupción más comunes en un computador de la familia 80 x 86 (que tenga como máximo 15 interrupciones).

### Tipos de interrupciones

Para clasificar los distintos tipos de interrupciones, se van a tener en cuenta dos criterios: dependiendo de la fuente que produce la interrupción y dependiendo del modo en que se obtiene el vector de interrupción.

En la tabla que se muestra a continuación se ve la clasificación según estos dos criterios.

Según la fuente de la interrupción		
Interrupciones hardware	Internas Son las producidas por la CPU.  Externas Producidas por los dispositivos E/S.	División por cero Desbordamiento Instrucción ilegal Dirección ilegal Logaritmo de cero Raíz cuadrada de números negativos Etc.  Vectorizadas No vectorizadas
Interrupciones software	Producidas por la ejecución de instrucciones de la CPU.	
Interrupciones autovectorizadas	El vector de interrupción es una posición de memoria asociada a la línea de interrupción, es fijo. 	
Interrupciones vectorizadas	El vector de interrupción (o una parte de él) es suministrado por el dispositivo. 	

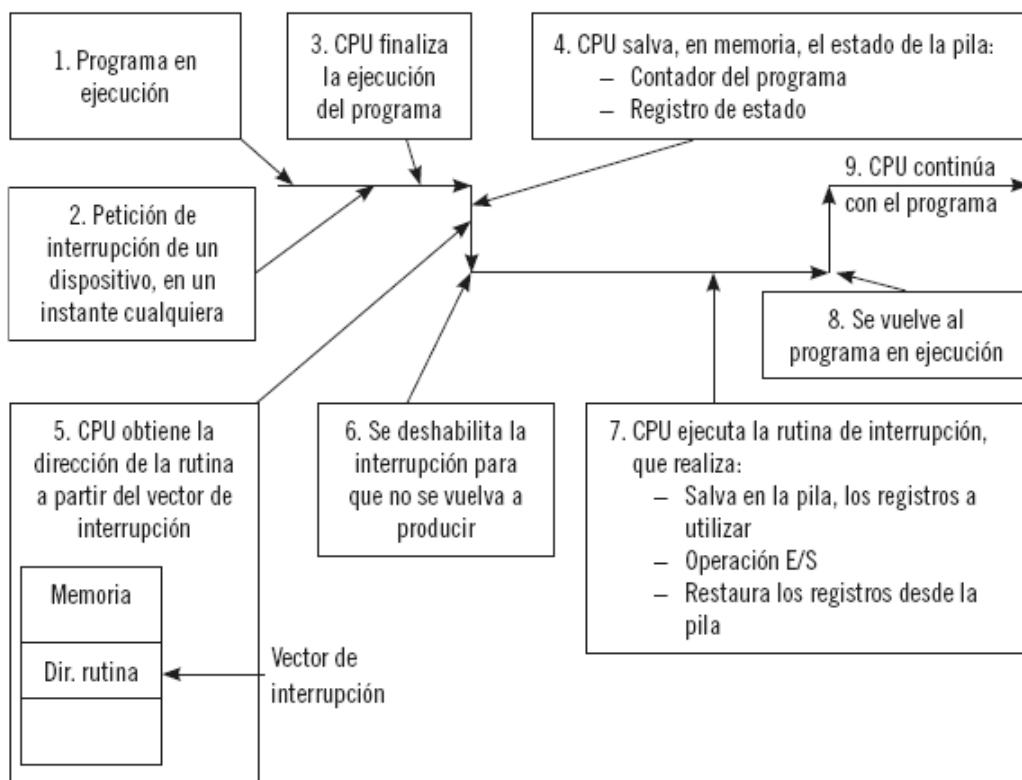
## Aplicación práctica

En un sistema se está ejecutando una aplicación de usuario. En un momento determinado se produce una operación de E/S que requiere la intervención del procesador.

Dibuje un gráfico en el que se detallen las fases que se producen en la ejecución de este código cuando se genera una interrupción. Describa cada una de ellas.

### Solución

El gráfico de las distintas fases que muestra qué hace un programa cuando llega una interrupción es el siguiente:



## Prioridades

El sistema de prioridades determina, cuando se produce una petición de interrupción de varios dispositivos de E/S de forma simultánea, cuál de dichas peticiones se va a ejecutar.

El orden de las prioridades se establece por *software*, en el caso de interrupciones de tipo no vectorizadas, ya que las peticiones van por una misma línea y el *hardware* en el caso de las interrupciones vectorizadas según la cercanía a la CPU.

### Enmascaramiento de interrupciones

En ocasiones, es necesario que no se produzca ninguna interrupción en la ejecución de un programa. Por esto, el sistema de interrupciones tiene la posibilidad de impedir que la CPU atienda interrupciones.

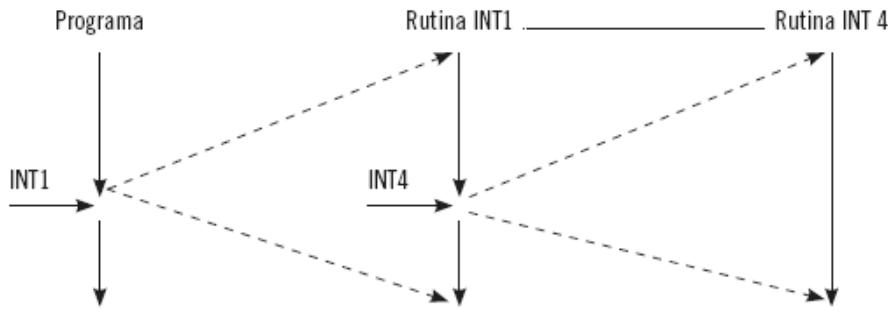
Desde el punto de vista del *hardware*, es posible deshabilitar solo determinadas interrupciones, utilizando un **registro de máscara**. Este registro se antepone a las señales que vienen de las distintas interrupciones permitiéndole continuar o no.

Desde el punto de vista del *software*, también se puede realizar este filtro. En el código del programa antes de llamar a la rutina de la interrupción, se identifica si interesa o no que se procese.

### Anidamiento de interrupciones

¿Es posible que se esté ejecutando una rutina de una interrupción y sea interrumpida por otra? La respuesta a esta pregunta es sí, y es posible gracias al sistema de prioridades, que determina que: "un dispositivo solo puede interrumpir a otro cuando su nivel de prioridad es mayor que el que se está atendiendo".

Ejemplo de llamadas anidadas de interrupciones



De forma práctica en los sistemas operativos de uso general *Windows* y *Linux*, existen formas de ver las interrupciones que están asociadas en el sistema.

Las interrupciones en *Windows* se pueden ver desde la pantalla de información del sistema.

En Linux se pueden ver las interrupciones en la carpeta: "/proc/" y dentro de ella en el fichero "interrupts". En este fichero pueden verse las interrupciones que están sin asociar, de manera que, si se asocia un nuevo dispositivo, se modificará el contenido del fichero.



### Nota

En Linux, "/proc/" no se trata de un sistema de archivos físico, se trata de un sistema virtual de archivos, que se utiliza para guardar información del sistema.

### Aplicación práctica

Se necesita controlar el envío de caracteres desde un dispositivo de entrada, en este caso un teclado, a un sistema *Windows*.

Para ello se va a comprobar cuál es la IRQ (*Interrupt ReQuest*) que se envía al procesador cuando se pulsa una tecla y que sirve para indicar que hay datos para procesar.

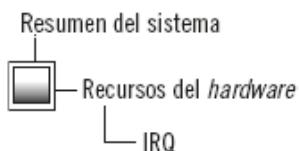
¿Cómo se ve la información de las IRQ en *Windows*?

¿Cuál IRQ es la del teclado?

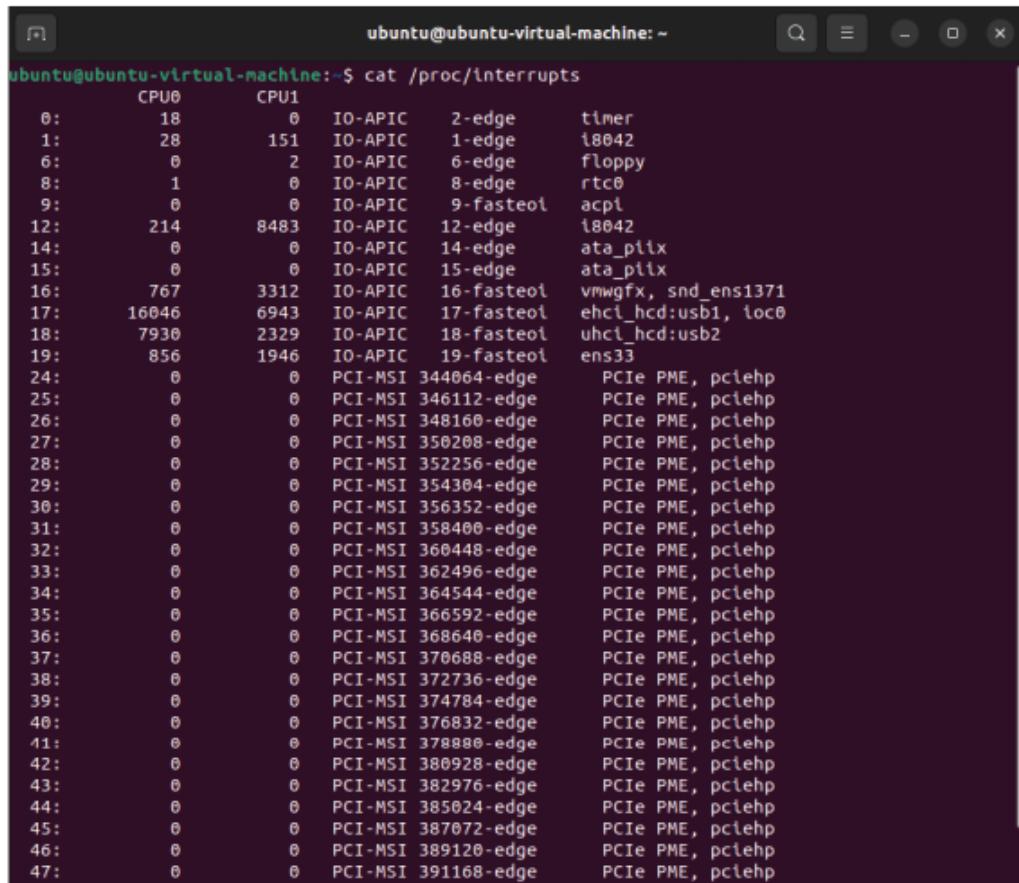
### Solución

Utilizando el comando "msinfo32.exe", en la consola de *Windows*, se accede a la información del sistema.

Aparecerá la pantalla con la información del sistema y en ella se navega por:



Marcando IRQ aparece en la parte derecha la información de los dispositivos que utilizan cada interrupción y enumera cuáles quedan libres, tal como aparece en la pantalla que se muestra a continuación.



```
ubuntu@ubuntu-virtual-machine:~$ cat /proc/interrupts
          CPU0      CPU1
 0:      18      0  IO-APIC   2-edge      timer
 1:      28     151  IO-APIC  1-edge      i8042
 6:      0       2  IO-APIC   6-edge    floppy
 8:      1       0  IO-APIC   8-edge      rtc0
 9:      0       0  IO-APIC   9-fasteoi  acpi
12:     214    8483  IO-APIC  12-edge      i8042
14:      0       0  IO-APIC  14-edge  ata_piix
15:      0       0  IO-APIC  15-edge  ata_piix
16:     767    3312  IO-APIC  16-fasteoi  vmwgfx, snd_ens1371
17:   16046    6943  IO-APIC  17-fasteoi  ehci_hcd:usb1, ioc0
18:   7930    2329  IO-APIC  18-fasteoi  uhci_hcd:usb2
19:    856    1946  IO-APIC  19-fasteoi  ens33
24:      0       0  PCI-MSI 344064-edge    PCIe PME, pciehp
25:      0       0  PCI-MSI 346112-edge    PCIe PME, pciehp
26:      0       0  PCI-MSI 348160-edge    PCIe PME, pciehp
27:      0       0  PCI-MSI 350208-edge    PCIe PME, pciehp
28:      0       0  PCI-MSI 352256-edge    PCIe PME, pciehp
29:      0       0  PCI-MSI 354304-edge    PCIe PME, pciehp
30:      0       0  PCI-MSI 356352-edge    PCIe PME, pciehp
31:      0       0  PCI-MSI 358400-edge    PCIe PME, pciehp
32:      0       0  PCI-MSI 360448-edge    PCIe PME, pciehp
33:      0       0  PCI-MSI 362496-edge    PCIe PME, pciehp
34:      0       0  PCI-MSI 364544-edge    PCIe PME, pciehp
35:      0       0  PCI-MSI 366592-edge    PCIe PME, pciehp
36:      0       0  PCI-MSI 368640-edge    PCIe PME, pciehp
37:      0       0  PCI-MSI 370688-edge    PCIe PME, pciehp
38:      0       0  PCI-MSI 372736-edge    PCIe PME, pciehp
39:      0       0  PCI-MSI 374784-edge    PCIe PME, pciehp
40:      0       0  PCI-MSI 376832-edge    PCIe PME, pciehp
41:      0       0  PCI-MSI 378880-edge    PCIe PME, pciehp
42:      0       0  PCI-MSI 380928-edge    PCIe PME, pciehp
43:      0       0  PCI-MSI 382976-edge    PCIe PME, pciehp
44:      0       0  PCI-MSI 385024-edge    PCIe PME, pciehp
45:      0       0  PCI-MSI 387072-edge    PCIe PME, pciehp
46:      0       0  PCI-MSI 389120-edge    PCIe PME, pciehp
47:      0       0  PCI-MSI 391168-edge    PCIe PME, pciehp
```

Pantalla de información del sistema en la que aparecen todas las IRQ (interrupciones) asignadas a dispositivos en un sistema Windows 7.

En la imagen se puede comprobar que la IRQ del teclado es la **IRQ 1**.



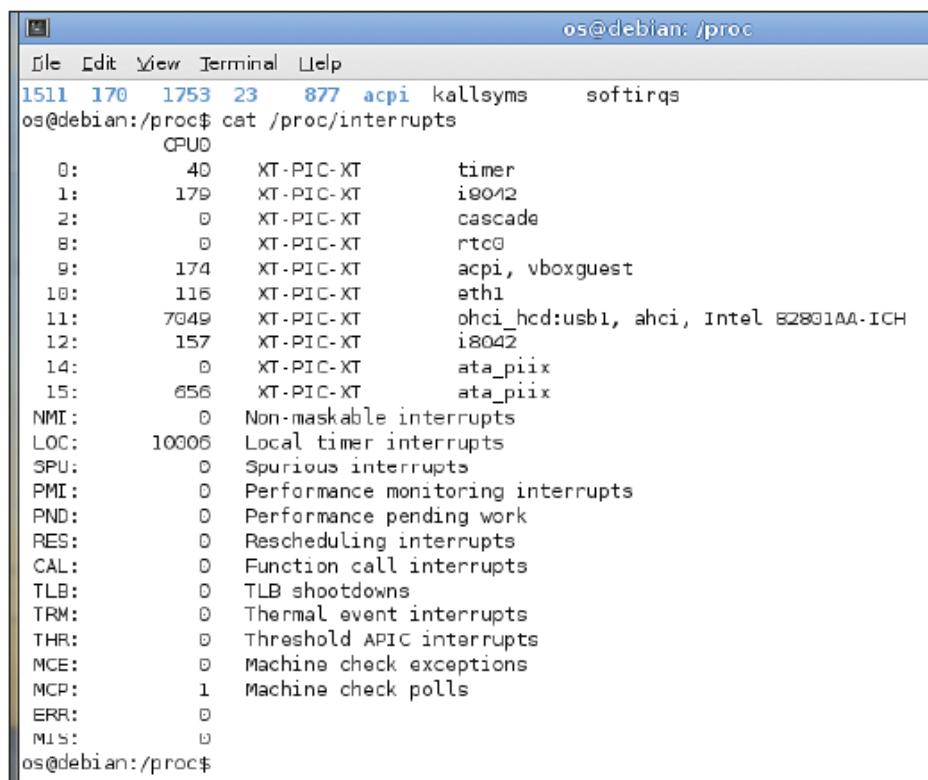
### Aplicación práctica

Se pretende programar un **driver** que tenga en cuenta las interrupciones que emite el reloj del sistema (**timer**). El **driver** se va a realizar para un sistema **Linux**. Se necesita saber cómo se puede acceder a las IRQ que están siendo utilizadas por el sistema y comprobar cuál es la IRQ que utiliza el reloj (**timer**).

### SOLUCIÓN

Para ver las interrupciones que están actualmente en uso por el sistema, se usa el comando “cat”, que muestra por pantalla el contenido de lo que se le envíe como argumento. En este caso, se le va a enviar como argumento /proc/interrupts.

Esto se hace como en otras ocasiones desde el terminal de *Linux*.



The screenshot shows a terminal window titled "os@debian: /proc". The window contains the output of the command "cat /proc/interrupts". The output is as follows:

```
1511 170 1753 23 877 acpi kallsyms      softirqs
os@debian:/proc$ cat /proc/interrupts
CPU0
 0:      40 XT-PIC-XT      timer
 1:    179 XT-PIC-XT      i8042
 2:      0 XT-PIC-XT      cascade
 8:      0 XT-PIC-XT      rtc@
 9:    174 XT-PIC-XT      acpi, vboxguest
10:   116 XT-PIC-XT      eth1
11: 7049 XT-PIC-XT      ohci_hcd:usb1, ahci, intel 82801AA-ICH
12:   157 XT-PIC-XT      i8042
14:     0 XT-PIC-XT      ata_piix
15:   656 XT-PIC-XT      ata_piix
NMI:     0 Non-maskable interrupts
LOC: 10006 Local timer interrupts
SPU:     0 Spurious interrupts
PMI:     0 Performance monitoring interrupts
PND:     0 Performance pending work
RES:     0 Rescheduling interrupts
CAL:     0 Function call interrupts
TLB:     0 TLB shootdowns
TRM:     0 Thermal event interrupts
THR:     0 Threshold APIC interrupts
MCE:     0 Machine check exceptions
MCP:     1 Machine check polls
ERR:     0
MIS:     0
os@debian:/proc$
```

Terminal de *Linux* que muestra las interrupciones en uso

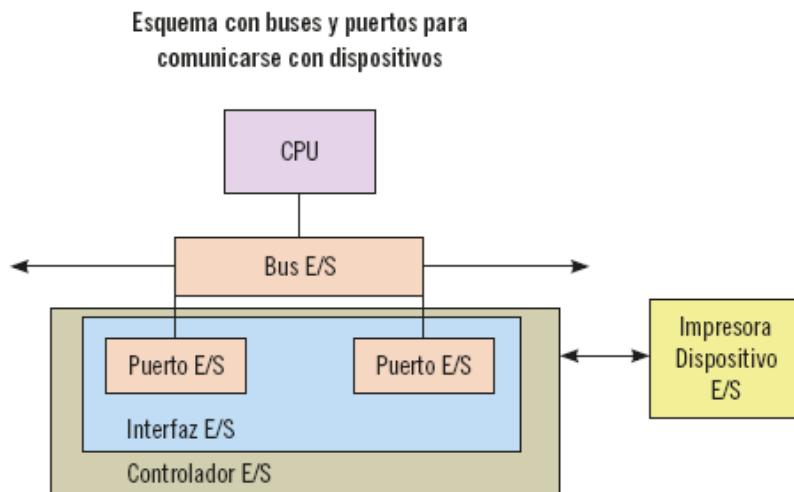
En la imagen se puede comprobar que la IRQ 0 (el número que aparece en la columna a la izquierda de la pantalla) es la usada por el timer.

## 4.5. Gestión de puertos de entrada y salida

El tránsito de información entre los distintos elementos del sistema (CPU, memoria, dispositivos de E/S) se realiza a través de un medio compartido para la comunicación, que se denomina **bus**. Cabe destacar que un dispositivo se conecta a un solo bus.

En la figura que se muestra a continuación, se ven los elementos que interfieren en la comunicación entre los dispositivos y la CPU:

- Los buses.
- Puertos de E/S.
- Una interfaz de comunicación.
- Un controlador de E/S.



Los dispositivos están conectados a través del bus y disponen de una serie de direcciones, llamadas **puertos** de E/S, que van a servir como enlace para que puedan comunicarse.



#### Sabía que...

El número de puertos que soporta la arquitectura de un PC es de 65.536 de 8 bits.

La comunicación entre la CPU y los dispositivos, a través de los puertos, se puede hacer de dos maneras:

- Mediante instrucciones directas a los puertos en ensamblador.
- Mapeando los puertos en memoria y accediendo a ellos a través de instrucciones de acceso directo a memoria. Esta es la opción que se utiliza, ya que es más cómoda y rápida.

Para realizar la comunicación, cada controlador E/S tiene una serie de registros que se utilizan para:

- Recibir o entregar datos.
- Conocer el estado del dispositivo con el que se establece la comunicación.
- Realizar operaciones específicas, del tipo encenderse o apagarse, etc.



Mediante el registro de control se pueden indicar operaciones específicas al dispositivo. El registro de estado informa de cómo se encuentra el dispositivo en ese momento. Mediante los registros de entrada y salida, se llevan a cabo el envío y la recepción de datos.

El acceso a un puerto de E/S no es una tarea complicada; lo que sí se complica es la asociación entre el dispositivo, al que se quiere acceder, y el puerto que tiene asociado. Un error que se da a menudo es que un controlador o *driver* intente escribir en un puerto que está siendo usado por otro dispositivo, con lo cual se produce un fallo del sistema.

Para solucionar este tema, el núcleo del sistema tiene una **lista de puertos E/S asignados**.



### Actividades

7. Busque una tabla que muestre las direcciones hexadecimales de los puertos de E/S más frecuentes.

De forma práctica, en los sistemas operativos de uso general *Windows* y *Linux*, existen formas de ver los puertos que están asociados en el sistema.

Los puertos en *Windows* se pueden ver desde la pantalla de información del sistema. En *Linux* se pueden ver los puertos en la carpeta: “/proc/” y dentro de ella en el fichero “iports”.



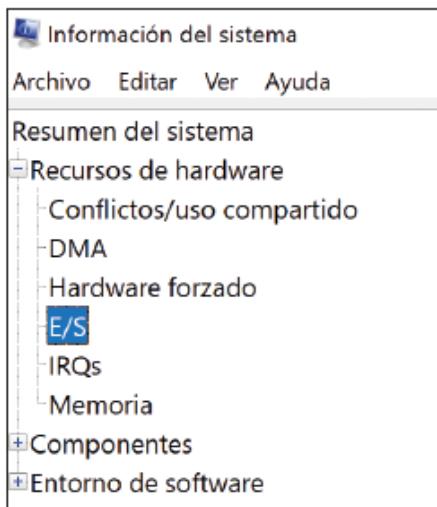
### Aplicación práctica

**Para programar los puertos que utiliza un driver, se precisa encontrar, en un sistema *Windows*, información de todos los puertos E/S de los que se dispone en el sistema.**

### SOLUCIÓN

Utilizando el comando “msinfo32.exe” en la consola de *Windows*, se accede a la información del sistema.

Aparecerá la pantalla con la información del sistema y en ella se navega por:



Marcando E/S aparece en la parte derecha la información de todos los puertos de E/S en uso y los dispositivos que los utilizan, tal como aparece en la pantalla que se muestra a continuación:

Información del sistema			
Resumen del sistema	Recurso	Dispositivo	Estado
Recursos de hardware	0x00000000-0x000000..	Bus PCI	OK
Conflictos/uso compartido	0x00000000-0x000000..	Controladora de acceso directo a memoria	OK
DMA	0x00000000-0x000000..	Recursos de la placa base	OK
Hardware forzado	0x00000010-0x000000..	Recursos de la placa base	OK
E/S	0x00000020-0x000000..	Controladora programable de interrupciones El..	OK
IRQs	0x00000024-0x000000..	Recursos de la placa base	OK
Memoria	0x00000028-0x000000..	Recursos de la placa base	OK
Componentes	0x0000002C-0x000000..	Recursos de la placa base	OK
Entorno de software	0x0000002E-0x000000..	Recursos de la placa base	OK
	0x00000030-0x000000..	Recursos de la placa base	OK
	0x00000034-0x000000..	Recursos de la placa base	OK
	0x00000038-0x000000..	Recursos de la placa base	OK
	0x0000003C-0x000000..	Recursos de la placa base	OK
	0x00000040-0x000000..	Temporizador del sistema	OK
	0x00000050-0x000000..	Recursos de la placa base	OK
	0x00000060-0x000000..	Teclado PS/2 estándar	OK
	0x00000061-0x000000..	Altavoz del sistema	OK
	0x00000064-0x000000..	Teclado PS/2 estándar	OK
	0x00000070-0x000000..	Sistema CMOS/reloj en tiempo real	OK
	0x00000072-0x000000..	Recursos de la placa base	OK
	0x00000080-0x000000..	Recursos de la placa base	OK
	0x00000081-0x000000..	Controladora de acceso directo a memoria	OK
	0x00000090-0x000000..	Recursos de la placa base	OK
	0x000000A0-0x000000..	Controladora programable de interrupciones El..	OK
	0x000000A4-0x000000..	Recursos de la placa base	OK
	0x000000A8-0x000000..	Recursos de la placa base	OK
	0x000000AC-0x000000..	Recursos de la placa base	OK
	0x000000B0-0x000000..	Recursos de la placa base	OK
	0x000000B8-0x000000..	Recursos de la placa base	OK
	0x000000BC-0x000000..	Recursos de la placa base	OK
	0x000000C0-0x000000..	Controladora de acceso directo a memoria	OK

Buscar esto:  Buscar  Cerrar búsqueda

Buscar solo la categoría seleccionada  Buscar solo nombres de categoría

Pantalla de información del sistema en la que aparecen todas las E/S asignadas de un sistema Windows 7

#### 4.6. Uso de Acceso Directo a Memoria (DMA) y buses

Tanto las E/S controladas por programación como las interrupciones tienen una gran desventaja y es que consumen demasiado tiempo de la CPU.

En el caso de las E/S programadas, la CPU está dedicada en exclusiva a la operación que esté realizando el dispositivo, y si se usan interrupciones, aunque liberan la CPU en mayor medida, es necesario añadir instrucciones ajenas a la transferencia que también necesitan de tiempo de CPU.

Para solucionar estos problemas se utiliza el **Acceso Directo a Memoria (DMA)**, que se puede considerar como un módulo que puede leer y escribir, en la memoria principal, los datos que están involucrados en las transferencias de los dispositivos de E/S. Libera por completo la CPU.

Este módulo DMA está en el bus y puede acceder directamente a la memoria como si se tratase de la CPU.

En la tabla que se muestra a continuación, puede verse un resumen de las distintas técnicas para transferencia de datos de E/S.

	<b>Sin interrupciones</b>	<b>Con interrupciones</b>
Transferencia E/S a memoria a través de la CPU	E/S programada	E/S mediante interrupciones
Transferencia directa de E/S a memoria		DMA

### Estructura de un bus

Como se ha visto anteriormente, el **bus** se encarga de transmitir la información entre las unidades. Si una unidad inicia y controla la transferencia, se la considera **maestro del bus (master)** y, si es la unidad la que recibe los datos, se la considera **esclavo (slave)** de la transferencia.

En el caso de la unidad DMA, como se verá a continuación al tratar de su funcionamiento, realiza el papel de esclavo, en relación con la transferencia que inicializa la CPU, para las operaciones de E/S, y también hace de maestro para la transferencia con la memoria.

Las líneas de un bus se pueden clasificar según el papel que cumplen durante las transferencias:

- **Líneas de información básica:** utilizadas por el maestro para definir los dos elementos básicos de una transferencia, el esclavo y los datos. De este tipo, se clasifican en:
  - Líneas de direcciones: indican la unidad esclavo.
  - Líneas de datos: llevan los datos a transferir.
- **Líneas de control:** a través de estas líneas se transmiten órdenes relacionadas con la operación E/S. De este tipo, se encuentran:
  - Escritura/lectura en memoria.
  - Operaciones de salida/entrada.
- **Líneas de arbitraje:** para establecer peticiones al bus, prioridades, ocupación y cesión del bus.

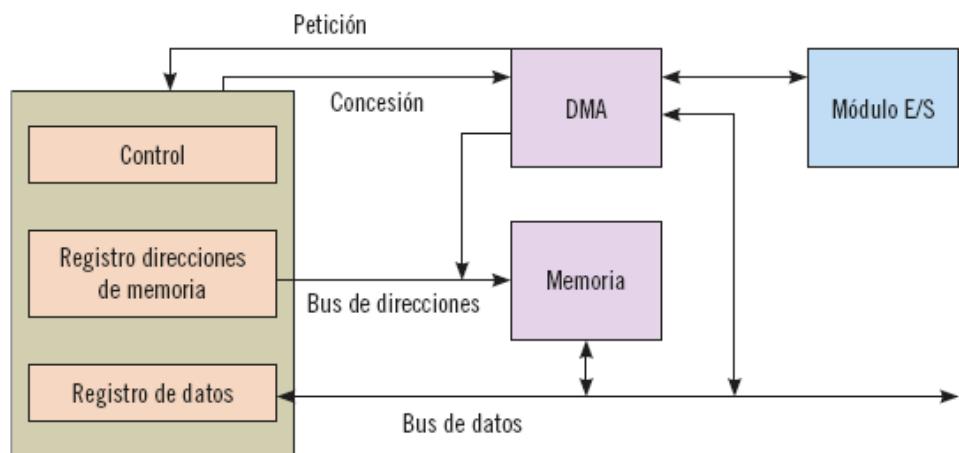
### Funcionamiento de la unidad DMA

Cuando se necesita realizar una operación de E/S con un dispositivo, ocurre lo siguiente:

- La CPU inicializa al DMA.
- El DMA solicita a la CPU la petición para el acceso a la memoria.
- La CPU prepara a los buses del sistema para la transmisión, la inicializa.
- El DMA accede a la memoria.
- Al finalizar la transferencia la CPU retoma el control.

La velocidad de transferencia está relacionada con el **ancho de banda de la memoria**.

### Esquema de la comunicación entre la CPU y el módulo E/S a través de DMA



### Definición

#### Ancho de banda de la memoria

Cantidad de datos que se pueden manejar (leer o almacenar) a la vez. Normalmente, viene expresado en unidades de bytes por segundo.



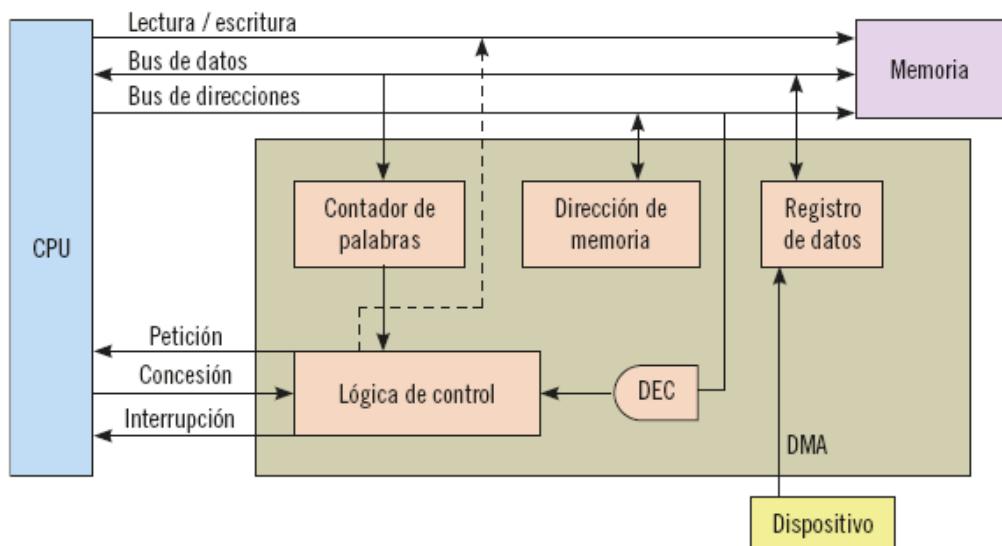
### Actividades

8. Busque información sobre el sistema de acceso ordenado que se establece en un bus, cuando varias unidades pretenden acceder al mismo tiempo, el llamado sistema de arbitraje. Este sistema garantiza que solo exista un maestro (master) en cada momento.

### Estructura de un controlador DMA

Un controlador DMA está compuesto por los elementos que aparecen en la siguiente imagen:

### Esquema de los componentes de un DMA



Los elementos son los siguientes:

- Registro de dirección: contiene la dirección de la siguiente palabra a transmitir.
- Contador de palabras: contiene el número de palabras que restan para transmitir.
- Lógica de control: es una circuitería que controla el contenido del contador de palabras y cuándo se llega a la última.
- Decodificador (DEC): identifica la dirección de memoria que le ha sido asignada al DMA.

Cuando se va a realizar una operación con un dispositivo, la CPU envía a la DMA lo siguiente:

- Tipo de operación: si es escritura o lectura.
- Dirección del dispositivo.
- Posición de memoria a partir de la cual comienza la lectura o la escritura del bloque.
- Número de palabras que estarán contenidas en el bloque.

Una vez que se ha enviado esta información, la CPU se despreocupa de esta tarea, puede continuar realizando otras operaciones y es el DMA el que transfiere toda la información entre el dispositivo y la memoria, sin interferir en la CPU. Al finalizar la tarea, el DMA envía una interrupción a la CPU.

## Tipos de control del bus con DMA

Mientras dura la transferencia del DMA, es esta unidad la que controla el bus y lo comparte con la CPU.

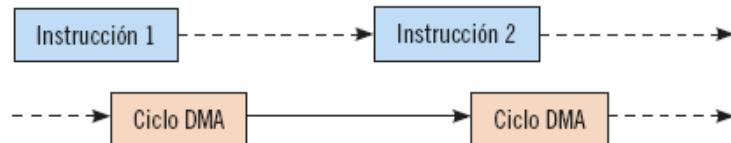
Existen distintas formas de compartir el bus y cada una de ellas ofrece unas características diferentes: mayor velocidad, mayor tiempo de actividad de la CPU, etc.

La elección de una alternativa u otra dependerá de las prestaciones que se deseen potenciar.

Los tipos de control son los siguientes:

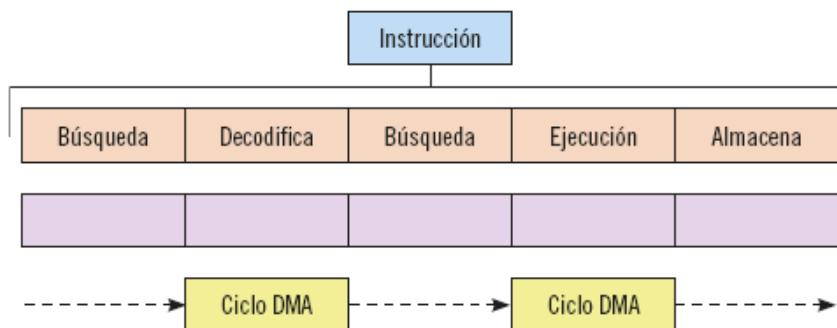
- **Ráfagas:** el control del bus, mientras dure la transmisión de un bloque de datos completo, va a recaer sobre la unidad DMA.
- **Robo de ciclos:** el DMA controla el bus durante un ciclo cada vez, transmite una palabra y espera a otro ciclo en que el bus esté disponible.

Control del bus tipo robo de ciclo



- **Transparente:** el acceso al bus por parte del DMA se realiza solo cuando la CPU no lo necesita. El programa que se esté ejecutando no se ve afectado en la velocidad de ejecución. En la ejecución de un programa hay fases, por ejemplo la de decodificación y la de ejecución, que no necesitan usar el bus, así que es en este momento cuando la unidad DMA hace uso de él.

Control del bus tipo transparente

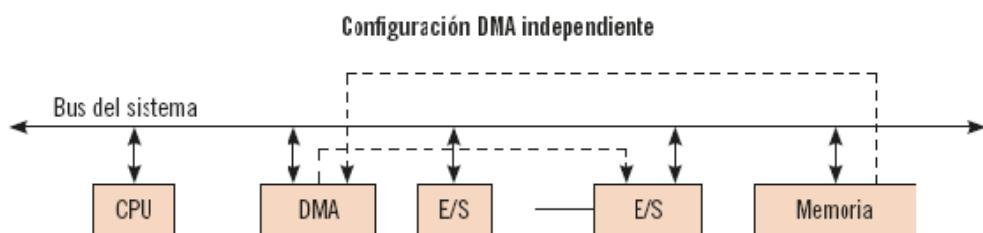


## Configuraciones de DMA

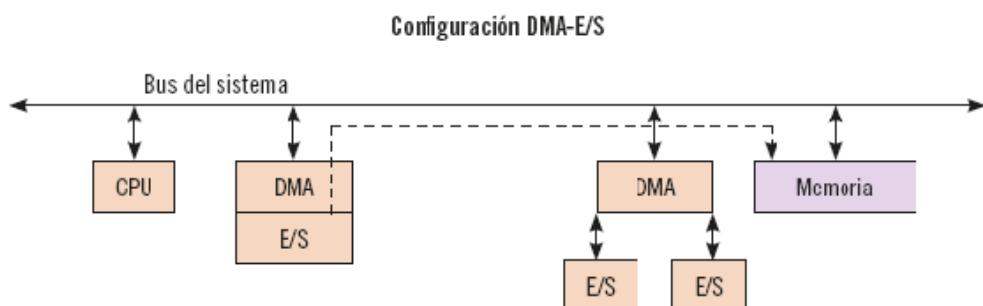
La configuración del DMA describe qué posiciones adopta este en relación al bus del sistema.

Se pueden distinguir las siguientes configuraciones:

- **DMA independiente (un único bus)** en esta configuración todos los módulos comparten el mismo bus del sistema, de manera que a la hora de la transferencia de datos el DMA actúa como intermediario entre la memoria y el dispositivo. La ventaja de esta configuración es que es económica, pero cada vez que se transfiere una palabra se consumen dos ciclos del bus, con lo cual **es ineficiente**.

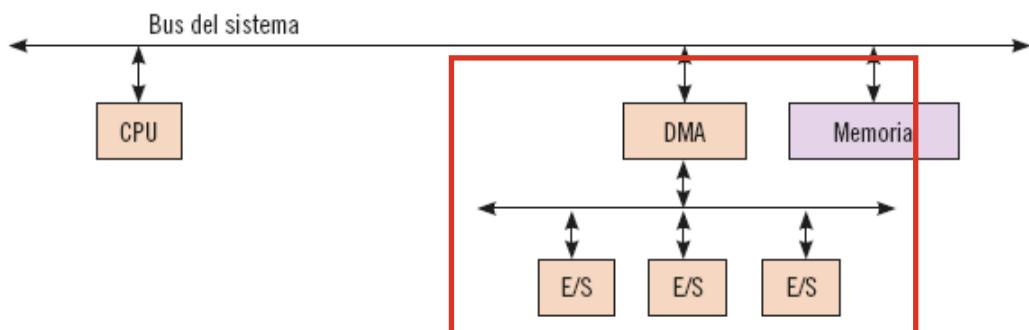


- **Integración de funciones DMA-E/S:** en esta configuración para uno o más controladores de dispositivos de E/S existe una conexión directa con el DMA, sin incluir el bus del sistema. **Se reduce a uno el número de ciclos** que se utilizan del bus del sistema.



- **Bus de E/S conectado al DMA**: una mejora sobre la configuración anterior que hace que **se reduzca el número de interfaces de E/S al DMA** es la configuración de bus de E/S conectado al DMA. En esta configuración, se va a utilizar un mismo bus para todos los controladores de los dispositivos de E/S al DMA. Además, con este tipo de configuración la ampliación es muy sencilla.

### Configuración bus de E/S conectado al DMA



### Actividades

9. Busque información sobre la evolución en la gestión de operaciones de E/S, los procesadores de E/S.



### Aplicación práctica

**Se va a realizar una transferencia de una palabra (byte), pero se necesita saber cuál será la opción que implicará menos a la CPU mientras se realiza la transferencia.**

**Las tres opciones que se pueden dar son:**

- | **TP: tiempo de E/S controlado por programa.**
- | **TI: tiempo de transferencia con interrupciones.**
- | **TD: tiempo de transferencia usando DMA.**

### SOLUCIÓN

La participación de la CPU es de menor a mayor la siguiente: TD < TI < TP.

Es decir, con DMA la CPU participa muy poco en relación con la E/S controlada por programa. Si se usan las interrupciones, la CPU participa cuando un dispositivo lo necesite.

## 5. Técnica de depuración y prueba

Antes de abordar la programación de un controlador de dispositivo, es necesario tener en cuenta varias cosas:

- **Conocer en profundidad el funcionamiento del dispositivo.** Para programar un *driver* que influya sobre el comportamiento de un dispositivo, hay que hacer un estudio exhaustivo de que es lo que hace este. Desde el punto de vista *hardware*, hay que saber cómo va a circular la información del dispositivo, además de qué protocolos utiliza para conectarse.
- **Analizar el problema.** Este es un punto importante, ya que hay que tener muy bien definido qué es lo que se espera del *driver*. Es necesario saber cuáles van a ser las funcionalidades que se necesitan, identificarlas y describirlas para que sea más fácil el desarrollo del código.
- **Definir las necesidades.** En esta fase se definen las funciones que se van a necesitar, las estructuras de datos a utilizar, etc., de forma detallada.
- **Diseñar los algoritmos.** Es en este momento cuando se define qué es lo que van a hacer las funciones identificadas en el paso anterior.
- **Programar el código del controlador.** En esta fase, partiendo de las especificaciones que se han realizado previamente, se pasa a la codificación o, lo que es lo mismo, a realizar el programa en el lenguaje que se haya elegido. Normalmente, será C/C++.
- **Depurar y verificar el código del programa.** Esta es la última fase antes de dar por válido el programa realizado y que pase a funcionar. Es necesario hacer pruebas efectivas para evitar que se produzcan errores una vez que esté el *driver* en funcionamiento.



### Definición

#### Algoritmo

Secuencia de instrucciones necesarias para llevar a cabo una tarea previamente definida. Los algoritmos son independientes del lenguaje de programación y de la máquina en que se ejecuten.

#### VARIABLES

Espacio reservado de memoria que puede cambiar de contenido a lo largo de la ejecución de un programa.

#### CONSTANTES

Datos que se introducen al principio de un programa con un valor que no va a variar durante la ejecución.

### Biblioteca o librería

Conjunto de funciones que están codificadas en un lenguaje de programación. Al invocar una librería, se pueden utilizar las funciones que tiene implementadas.

Uno de los problemas más importantes a la hora de escribir el código de un *driver*, es decir, un código en modo *kernel*, es cómo se va a poder realizar la depuración del mismo. Un código *kernel* es difícil de depurar o trazar, ya que utiliza un conjunto de funcionalidades que no están asociadas a ningún proceso específico.

Los códigos de errores del *kernel* son también difíciles de reproducir e incluso, cuando se producen, pueden hacer que se colapse todo el sistema y con esto perder la información que llevó al error.

A continuación, se va a ver qué se puede hacer para depurar y rastrear los errores que se generen en el código de un *driver*.

## 5.1. Impresión de trazas

La técnica más común para depurar el código es realizando una supervisión en los puntos más interesantes de este.

Esta supervisión se hace imprimiendo, normalmente en un fichero de tipo *log*, alguna información que haga ver que el código va por buen camino.



### Nota

Un fichero de tipo *log* es un fichero plano, escrito por el núcleo, los servicios o las aplicaciones.

Ejemplos de información que puede servir son:

- El contenido de las variables implicadas
- Un mensaje que indique que ha pasado con éxito por un bucle, etc.

Lo que se envíe a imprimir depende del código que se esté utilizando.

La impresión de trazas ofrece la posibilidad de realizar un seguimiento exhaustivo del código, que resulta interesante para la búsqueda de errores (*debugging*), para el mantenimiento y para comprender el flujo de ejecución de dicho código.

Este sistema consiste en controlar mediante listados, pausas u otros mecanismos, los valores que tienen las variables a lo largo de la ejecución del programa.

Se puede hacer una traza de todo el programa o solo de una función determinada.

La forma más sencilla de realizar esta técnica es mediante la función de C:

- **printf**, que se encuentra implementada en la librería.
- **#include <stdio.h>** /\* invoca a la librería estándar de C: stdio.h \*/



#### Nota

---

En un programa C/C++, la palabra reservada `#include` invoca a una librería.

Las llamadas a librerías se encuentran ubicadas al principio del programa.

---

Por ejemplo: `printf("imprimo lo que interesa");` /\*Con esta función se imprime el dato del que se necesite hacer un seguimiento o un mensaje que indique que la ejecución va bien\*/.

## 5.2. Monitorización de errores

Aunque se haya utilizado una técnica para la depuración del código del *driver*, es posible que aparezcan errores y como consecuencia se produzca un fallo del sistema al ejecutar el *driver*.

Con la monitorización de los errores, se pretende conseguir lo siguiente:

- Identificar qué errores se producen en el código del *driver*.
- Definir qué acciones se van a realizar para tratar de solventarlos.

La gestión de los errores y la identificación de las situaciones anómalas es una de las partes más complicadas del desarrollo de *software* en general.

En el desarrollo de *drivers* en C/C++, las funciones pueden seguir una de estas formas de actuar en caso de que se detecte un error:

- **Terminar el programa.** Por ejemplo utilizando las funciones *exit* o *abort*. Esta no es una solución muy buena, ya que no ofrece ninguna alternativa en el caso de que se produzca un error.
- **Terminar el programa mediante la macro assert.** Esta macro permite abortar el programa cuando una determinada condición es falsa.
- **Devolver a la función un código de error.** Por ejemplo que una función devuelva un valor negativo cuando debe devolver siempre valores positivos, con esto se puede ayudar a detectar un error concreto.
- **Utilizar una variable global para almacenar el código de un error detectado.** De esta manera estará disponible para el resto del programa. Esta es una solución muy extendida. La biblioteca estándar de C introduce la **variable global errno**.
- **Lanzar una excepción.** Una excepción es un objeto, de cualquier tipo, que cuando se lanza detiene la ejecución de una función. El lanzamiento de la excepción se hace con *throw*. Este objeto va a saltar hasta encontrar un bloque que gestione las excepciones.

Para crear un bloque dentro del programa se utiliza *try* y para atrapar una excepción se utiliza *catch*.

Un ejemplo de utilización sería:

```
try
{
    //si situación anormal lanza excepción con throw
    // excepción

    ..
}

catch (tipo_1 iden_1)
{
    // las excepciones del tipo tipo_1
    //se gestionan a través del nombre iden_1
}

        catch (tipo_2 iden_2)
{
    // las excepciones del tipo tipo_2
    //se gestionan a través del nombre iden_2
}

catch (...)
{
    // cualquier otra excepción
    //se gestionan aquí
}
```

De esta manera, los errores quedarían identificados y se ofrecerían unas acciones para que no se produzca un colapso del sistema.

En definitiva, lo que se pretende es que no ocurra ningún suceso sin que quede identificado como un error y se pueda dirigir hacia una rutina que informe y actúe según la naturaleza del error.



Definición

### **Macro**

Instrucción escrita en un lenguaje de alto nivel que equivale a una serie de instrucciones en lenguaje máquina.

---



### **Actividades**

10. Busque información sobre las tablas que describen los códigos de error y sus descripciones para los sistemas *Linux* y *Windows*.
- 

## **5.3. Técnicas específicas de depuración de controladores en sistemas operativos de uso común**

En el diseño de un programa, es normal cometer errores (*bugs*) y por lo tanto es imprescindible una técnica para depurarlos (*debugging*).

En este apartado se va a ver cómo se aplican estas técnicas de depuración o *debugging*, de forma específica, para los sistemas operativos principales, es decir, las herramientas y las técnicas que aprovechan las particularidades de estos sistemas operativos.

### **Windows**

Para poner en práctica las técnicas de depuración que se van a utilizar en el sistema operativo *Windows*, se emplean una serie de programas e instrucciones propias. Esto va a permitir que el nuevo controlador a programar pueda comunicarse con las características propias de este sistema operativo.

*Windows* ofrece unas herramientas de depuración para los controladores, aplicaciones y servicios. El núcleo del motor de depuración de estas herramientas se llama *Windows debugger*.

Dentro de las herramientas de depuración, es posible distinguir las propias de *Windows* como *WinDbg*, y otras alternativas como *x64dbg*, y las que se utilizan a partir de las nuevas versiones de *Windows*, que forman parte de *Visual Studio*.



### **Nota**

---

Visual Studio es un entorno de programación integrado, para Windows, que soporta varios lenguajes de programación: C++, C#, Visual Basic, .NET, Java, etc.

Ofrece ediciones express, que son gratuitas, orientadas a estudiantes y a programadores júnior.

Estas versiones son iguales que las comerciales excepto por las herramientas avanzadas de integración, que no las incorporan.

---

### **Depuración por impresión**

La depuración por impresión consiste en el mismo sistema que se ha comentado anteriormente en la impresión de trazas. Pero en este entorno, en lugar de la función printf de C/C++, se va a utilizar otra función similar: **DbgPrint**.

El objetivo de esta función es enviar mensajes al depurador *kernel*. Para ello, va a recibir dos parámetros:

- Un parámetro en el que se le especifica el formato de la cadena a imprimir.
- La cadena de caracteres requerida. Esta cadena en muchas ocasiones incluirá el contenido de alguna variable.

La función DbgPrint está dentro de la librería **Ntddk.h**. Por eso, al programar un controlador para *Windows*, es necesario invocarla al principio del código.

Dentro del código del *driver*, se llamará a esta función donde sea más conveniente para hacer el seguimiento de la ejecución.



#### **Importante**

En los sistemas *Windows*, para poder ver los mensajes que se envían y poder realizar la depuración del código, se necesita una herramienta.

Con esta herramienta se van a poder ver los mensajes que se utilizan como argumentos de tipo cadena en la función DbgPrint.

Esta herramienta es **DebugView**, que, además de trabajar en modo kernel, puede hacerlo también en modo usuario.

---

Una vez realizada la programación del *driver*, es necesario compilar el código e instalarlo. A partir de ahí se pasa a la fase de pruebas.



## Definición

### Compilación

Consiste en el paso de un lenguaje de alto nivel a un lenguaje binario entendible por la máquina.

En esta etapa se va a utilizar la herramienta *DebugView*.

En esta ocasión, habrá que habilitar la opción modo *kernel*, para que sean capturados los mensajes del *kernel* de depuración. Después se mostrará la traza que ha generado el código.



### Importante

Para evitar que la programación de un *driver* provoque el colapso del sistema en el que se está trabajando, se recomienda que la instalación del nuevo *driver* se realice en una máquina virtual.

Una máquina virtual es un *software* que simula ser una máquina distinta dentro de un sistema. Se puede utilizar el mismo sistema operativo o instalar uno diferente. A todos los efectos es como estar trabajando en otro equipo.



### Actividades

11. Busque información sobre la herramienta para poder analizar la memoria del *kernel* de los sistemas *Windows*, *WinDbg*.



Uno de los grandes retos a los que se enfrentan los programadores de código para el *kernel* de *Unix/Linux* es la depuración, ya que este código no se ejecuta fácilmente en un depurador o *debugger*.

Existen varias técnicas de depuración: depuración por impresión, por consulta, por observación o por fallo del sistema, aunque la mayor parte de la depuración en *Unix/Linux* se hace por impresión.

En este apartado se van a describir estas técnicas, aunque incidiendo con más detalle en la de impresión.

### **Depuración por impresión**

Al igual que se ha comentado para los sistemas *Windows*, en *Linux* también se va a utilizar la impresión de trazas, pero utilizando una función más específica del núcleo para *Linux*.

En lugar de la función `printf`, se utilizará la función `printk`, que está implementada en la librería `<linux/kernel.h>`.

Esta es una función parecida a `printf`, pero con alguna peculiaridad más, ya que permite clasificar el mensaje de acuerdo con el nivel de gravedad que se considere.

En los ejemplos que aparecen a continuación, se van a mostrar dos mensajes, un mensaje de depuración y otro crítico:

- `printk(KERN_DEBUG "vamos por aquí %s:%i\n", __FILE__, __LINE__);`
- `printk(KERN_CRIT "por aquí vamos mal %p\n", ptr);`

El argumento que aparece en primera posición va a indicar la gravedad del mensaje que se quiere mostrar. Hay definidos hasta ocho niveles para indicar la criticidad del mensaje.



### **Actividades**

- 
12. Busque información sobre los mensajes que ofrece el *kernel* de *Linux* para utilizarlos como mensajes de depuración.

---

### **Depuración por consulta**

Debido a que la anterior técnica tiene como desventaja que penaliza la velocidad del sistema.

Esta técnica se basa en el uso del sistema de ficheros de `"/proc/"` que permite leer los datos del *kernel*.

El controlador escribirá en un fichero situado en `"/proc/"` para informar de cómo se está produciendo su ejecución.

### **Depuración por observación**

Con esta técnica se trata de observar el comportamiento de los programas, para lo que se utiliza el comando **strace**.

Este comando es una herramienta poderosa que permite mostrar todas las llamadas al sistema emitidas por un programa de usuario. Además, permite observar los argumentos a estas llamadas y los valores que se devuelven.

Cuando se produce un error, muestra el identificador del error (por ejemplo "ENOMEM") y la cadena que informa del tipo de error (en el ejemplo, "Fuera de memoria").

La información que muestra este comando es muy valiosa para los programadores del *kernel*.

### **Depuración por fallo del sistema**

Aunque se usen otras técnicas para la depuración del controlador, algunos errores pueden seguir produciéndose y generan un fallo del sistema cada vez que se ejecuta el controlador.

Lo importante en estos casos es recoger toda la información posible para analizarla y dar con el problema.

Como el código de *Unix/Linux* es muy robusto, si el controlador falla, lo más probable es que solo cierre este proceso y no todo el sistema. Cada fallo genera un mensaje informativo que se imprime en la consola y que puede servir para encontrar el error.

## **5.4. Aplicación de estándares de calidad del software de desarrollo al desarrollo de controladores de dispositivos**

Los estándares de calidad pueden definirse como unas normas a seguir para que un producto, ya sea *software*, servicios u otro tipo de producto, cumpla eficientemente las expectativas de los clientes/usuarios a los que va dirigido.

La aplicación de los estándares de calidad de *software* de desarrollo, en general, persigue una serie de objetivos que se pueden aplicar al caso concreto del desarrollo de controladores de dispositivos.

Los objetivos a perseguir son los siguientes:

- **Funcionalidad:** el *software* del controlador a desarrollar debe ser práctico y útil.

- **Confiabilidad:** mientras esté en funcionamiento, se busca que sea tolerante a fallos y que sea posible su recuperación, en el caso de que estos se produzcan, de forma fácil y rápida.
- **Usabilidad:** el controlador debe ser fácil de usar, es decir, que su manejo no sea difícil de aprender, y además debe ser operativo.
- **Eficiencia:** el controlador debe usar de forma óptima los recursos del sistema, controlando el tiempo de uso y los recursos que se utilicen.
- **Facilidad de mantenimiento:** el controlador debe estar preparado para ser modificado de forma cómoda, de cara a su mantenimiento.

Dentro de los modelos que abordan la calidad del *software*, el **modelo** más utilizado es el que se basa en el **proceso**. Algunos de los modelos basados en los procesos son: las normas ISO 9000 y las familias normativas ISO/IEC 15504 y 25000.



#### Sabía que...

ISO 9000 es una familia de estándares que se sigue en más de 80 países.

Este modelo persigue el análisis de las actividades del proceso de desarrollo del *software* que influyen en la calidad del producto.

En el desarrollo del proceso de un controlador de dispositivo, sería necesario seguir los siguientes pasos para obtener un *software* de calidad.

#### Requerimientos del software

En esta fase se va a plasmar, por escrito, cuál es el comportamiento que se quiere obtener una vez que se haya desarrollado el controlador. Para esto se crean los requerimientos que el *driver* debe cumplir.

Los requerimientos pueden ser:

- **Funcionales:** este tipo de requerimientos se refieren a qué funciones tiene que realizar el controlador.
- **No funcionales:** aquellos requerimientos que están relacionados con el rendimiento del sistema, interfaces de usuario, la robustez del desarrollo, etc.



#### Ejemplo

Se supone que la compañía X necesita crear un sencillo *driver*, "Hola, mundo", para *Linux*.

El objetivo del *driver* será que se escriba "Hola mundo" y "Adiós, mundo" al cargarse y descargarse el *driver*.

En este ejemplo, se va a realizar la fase de requerimientos de este *driver*.

La solución para este supuesto sería la siguiente:

En la fase de requerimiento del *software*, se realizaría un documento indicando por qué se necesita el nuevo *driver* y qué es lo que se quiere de él.

Podría ser así:

"Se ha detectado en el entorno en el que trabaja la compañía 'X', que se trata de un sistema operativo bajo *Linux*, la necesidad de disponer de un *driver* que envíe un mensaje a la hora de cargarse y otro cuando se haya descargado.

Los requisitos que debe cumplir el *driver* son los siguientes:

- Que se ejecute bajo *Linux*.
- Que se ejecute en modo kernel.
- Indicar mediante un mensaje que se ha producido la carga del *driver*.
- Indicar mediante un mensaje que se ha producido la descarga del *driver*.
- La compatibilidad con el sistema es un requisito indispensable".

De esta manera, quedarían expresados todos los requerimientos que se piden a este nuevo controlador.

---



### Aplicación práctica

**La empresa X, que ha solicitado el diseño del *driver* "Hola, mundo", necesita determinar con los requerimientos obtenidos cuáles pertenecen a requerimientos funcionales y cuáles no. La tabla de requerimientos es la siguiente:**

Requerimiento	Tipo
Realizar bajo <i>Linux</i>	
Realizar en modo <i>kernel</i>	
Mensaje de carga	
Mensaje de descarga	
Compatible sistema	

## SOLUCIÓN

Para clasificar los requerimientos, es necesario distinguir entre aquellos que especifican la función a realizar por el sistema, funcionales, y los que están relacionados con el rendimiento y otras cuestiones, no funcionales. Por lo tanto, la tabla quedaría de la siguiente manera:

Requerimiento	Tipo
Realizar bajo <i>Linux</i>	No funcional
Realizar en modo <i>kernel</i>	No funcional
Mensaje de carga	Funcional
Mensaje de descarga	Funcional
Compatible sistema	No funcional

## Diseño del software

En la fase de diseño se especifica de forma detallada cómo se van a solucionar los requerimientos que se han obtenido en la anterior fase. Todavía no se trabaja en ningún lenguaje de programación, se utiliza un lenguaje natural o bien pseudocódigo.



Definición

### Pseudocódigo

Descripción en un lenguaje informal de las operaciones que tiene que realizar un programa o algoritmo. Puede utilizar las estructuras de un lenguaje de programación, aunque está orientado al lenguaje natural.

La idea es pasar de los requerimientos obtenidos a cómo se van a solucionar. Esto se realiza mediante la abstracción de dichos requerimientos, es decir, del problema o requerimiento se pasa a una solución concreta.



### Ejemplo

En este ejemplo se va a realizar la fase de diseño del ejemplo visto anteriormente. Por lo tanto, se parte de los requerimientos obtenidos para el *driver* "Hola, mundo".

Se realizaría un documento indicando las funcionalidades detectadas en la fase de requerimientos y detallando sus posibles soluciones. Sería de la siguiente manera:

Para los requerimientos detectados, se proponen las siguientes soluciones:

- | Ejecutarse bajo *Linux*.
- | El *kernel* de *Linux* mediante una serie de funciones en el espacio de usuario permite que las aplicaciones puedan interactuar con el *hardware*.
- | En este sistema se hace con funciones de lectura y escritura de ficheros, ya que es así como *Linux* ve a los dispositivos. Los dispositivos estarán en el directorio \dev.
- | Por lo tanto, se va a implementar el nuevo *driver* en C/C++ con este tipo de funciones.
- | Que se ejecute en modo *kernel*.
- | Para que se ejecute en modo kernel se desarrollará y cargará en el sistema como un módulo *kernel*.
- | Indicar mediante un mensaje que se ha producido la carga del *driver*.
- | Es necesario realizar unas tareas para inicializar el dispositivo, como son reservar memoria, interrupciones, puertos E/S, etc. En esta función de inicialización, se va a imprimir el mensaje "Hola, mundo".
- | Indicar mediante un mensaje que se ha producido la descarga del *driver*.
- | En la función necesaria para descargar el módulo se va a imprimir un mensaje: "Adiós, mundo".
- | La compatibilidad con el sistema es un requisito indispensable.
- | Siguiendo el diseño de los anteriores requerimientos se garantiza que se cumpla este último.

De esta manera, queda definido el diseño para los requerimientos generados y están preparados para la implementación.

## Implementación del diseño

En esta fase es donde se codifica el controlador, utilizando el lenguaje de programación que se haya elegido, en este caso C/C++.

A partir de las soluciones que se han obtenido en la fase de diseño, se programan las funciones que van a formar parte del desarrollo del controlador.

Las funciones deben estar comentadas al detalle para que sean fácilmente modificables en la parte de mantenimiento.

Además, es muy importante que se tenga un cuidado especial a la hora de programar el tratamiento de errores.

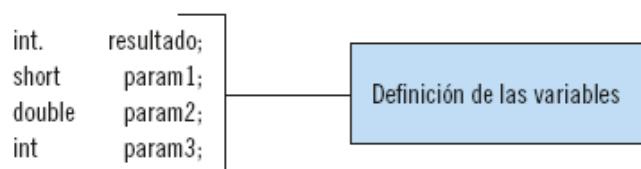
El uso de funciones proporciona una forma de utilizar un conjunto de instrucciones asociándolas a un nombre. Estas instrucciones se ejecutarán cada vez que este nombre sea invocado.

En C/C++, se definen de esta manera:

<b>Funciones sin parámetros</b>	Se define de la siguiente manera: en primer lugar el tipo de datos que devuelve, a continuación el nombre de la función y "()". Ejemplos: void mi_funcion(), --no devuelve nada int mi_funcion(), --devuelve un entero
<b>Funciones con parámetros</b>	Se define igual que la anterior, pero dentro de los paréntesis van los parámetros, separados por comas. Ejemplo: int mi_funcion (short parametro1,double parámetro2,int parámetro3);

Para ver cómo sería la llamada, dentro del código de un programa, a una función con parámetros, se vuelve al último ejemplo de la función "mi\_funcion".

En este caso, la función devuelve un entero y tendría tres parámetros. Primero se definen las variables que van a servir de argumentos de la función.



Una vez definidos y asignados los valores a estas variables, la llamada a la función sería de la siguiente manera:

```
resultado= mi_funcion(param1,param2,param3);
```

A la hora de implementar *drivers*, existen funciones que tienen que ir de forma obligada. Por ejemplo: es necesaria una función para realizar la carga del *driver* y otra para descargarlo.

En el caso de la programación de *drivers* en *Linux*, sería necesario implementar las funciones: **init\_module**, para la carga y **cleanup\_module** para la descarga.



### Ejemplo

En el ejemplo del *driver* "Hola mundo", se pasa a implementar las funciones una vez realizadas las fases de toma de requerimientos y diseño. Para ello, se va a realizar un pequeño código en lenguaje C/C++, para implementar las funciones **init\_module**, para la carga, y **cleanup\_module**, para la descarga.

El fichero en el que va a consistir el *driver* del ejemplo se va a llamar: "holamundo.c".

En la cabecera del fichero se llama a la librería que corresponde, en este caso "module.h".

```
<<holamundo.c>>=
#define MODULE
#include <linux/module.h>
```

Para la función de carga, la función que se va a utilizar para detectar que se ha cargado el módulo es:

```
int init_module(void)
{
    printk("<1>Hola mundo\n");
    return 0;
}
```

Se ha utilizado la función printk porque es similar a **printf** y solo puede ser utilizada en el núcleo. El primer argumento de la función es <1>, que indica el nivel de criticidad del mensaje. En este caso indica que es de alta prioridad. Con esto se consigue que de forma obligatoria lo muestre en pantalla.

Para la función de **descarga**, la función que se va a utilizar para descargar el módulo es:

```
void cleanup_module(void)
{
    printk("<1>Adios mundo\n");
}
```

Y cuando se descargue el módulo mostrará el mensaje.

El sencillo *driver* queda implementado de esta manera.

---



### Actividades

13. Busque información sobre la función para C/C++ utilizada en el *kernel de Linux*, "printk", vea qué tipo de parámetros acepta y qué devuelve.
  14. Amplíe información sobre cómo se definen las funciones en el lenguaje de programación C/C++ y las distintas formas de paso de variables
- 

### Integración y prueba del software

El buen funcionamiento de las funciones programadas y la integración de todas ellas, de forma que el comportamiento sea el esperado, es el objetivo principal de esta fase.

Es muy importante la realización de una batería de pruebas, a cada una de las funciones y al conjunto, de cara a minimizar los errores que puedan aparecer.

### Integración y prueba del sistema

En esta fase se van a realizar las pruebas del controlador dentro del sistema y, por lo tanto, es una fase delicada.

Un fallo no controlado anteriormente puede traducirse en un error del sistema.

Hay que estar preparado para restaurar el sistema en caso de que se llegue a este punto. Por ello, es recomendable hacer una copia del sistema previo a esta fase, con el fin de minimizar riesgos de pérdida de información.

### Mantenimiento del software

Una parte importante de esta fase es la realización de un manual detallado de la instalación y uso del controlador, de cara a los usuarios.

También es necesario realizar un manual técnico, en el que se describa el desarrollo del controlador, para futuras modificaciones. Esto hace que sea más fácil su mantenimiento más adelante, tanto para el propio desarrollador como para otros.



### Actividades

15. Busque información sobre la metodología Métrica 3, que sistematiza actividades dentro del ciclo de vida del *software*. Puede verse en el portal de la administración electrónica dependiente de la Secretaría General de Administración Digital del Ministerio de Asuntos Económicos y Transformación Digital.

## 6. Compilación y carga de controladores de dispositivos

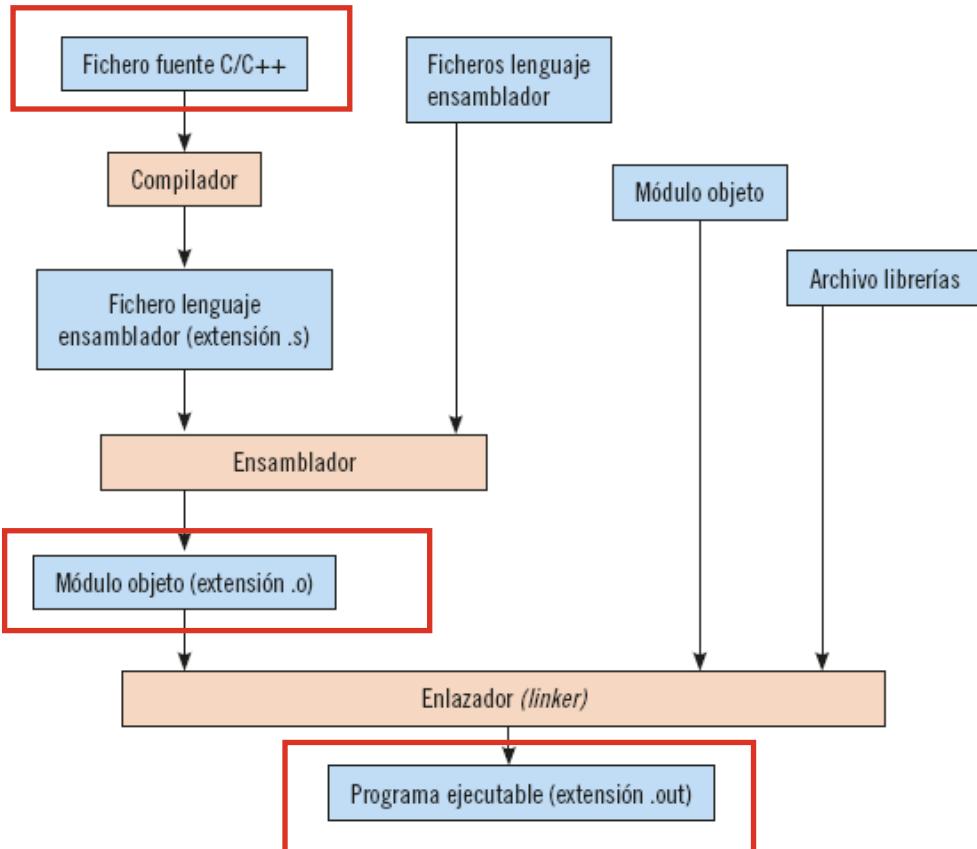
Los controladores de dispositivos escritos en C/C++ están expresados en un **lenguaje de alto nivel**, es decir, un lenguaje que está más cerca del lenguaje humano.

Para que el sistema entienda este *software*, es necesario acercarlo a un **lenguaje máquina**, y esto es lo que se hace al compilar el programa, en este caso el *driver*.

En la imagen que aparece a continuación se ve la evolución y las herramientas que intervienen desde que un código se escribe en un lenguaje de programación hasta que pueda ser entendido por el sistema para que pueda influir sobre el *hardware*.

Un controlador está compuesto por un conjunto de rutinas que son llamadas por el sistema operativo. Las rutinas pueden ser llamadas en las siguientes situaciones:

### Flujo de ficheros fuente a través de la cadena de herramientas



- En la carga y descarga del controlador.
- Al conectar o desconectar un dispositivo al sistema.
- En las peticiones de E/S que realiza el usuario, etc.

De forma más concreta, los eventos que se van a tener en cuenta de forma general por los *drivers* son:

- Carga de módulo.
- Abrir dispositivo.
- Leer dispositivo.
- Escribir dispositivo.
- Cerrar dispositivo.
- Descargar/quitar el módulo.

En esta sección, se va a describir cómo se puede compilar un *driver*, además de cómo se van a producir dos de los eventos enumerados anteriormente, la carga y la descarga de un módulo, en los dos sistemas operativos de uso común.



## Aplicación práctica

En un directorio están almacenados los archivos que se listan a continuación. ¿En qué orden se han generado los archivos?, ¿qué herramientas se han utilizado para la generación del archivo?

**La lista de archivos es la siguiente:**

**new\_prog.s; new\_prog.out; new\_prog.o; new\_prog.c.**

### SOLUCIÓN

Para ver de forma gráfica el orden en que se han generado los archivos y qué herramientas se han utilizado para generarlos con esa extensión, se usará la siguiente tabla:

Fichero	Herramienta
new_prog.c	Programa para generar código en alto nivel C/C++
new_prog.s	Compilador
new_prog.o	Ensamblador
new_prog.out	Linker

## Windows

Para desarrollar *drivers* en Windows es necesario utilizar un entorno especial proporcionado por Microsoft (**WDK**), que se verá más adelante.

Este entorno de programación ofrece además la posibilidad de compilar los nuevos *drivers*, teniendo en cuenta la versión de Windows a la que van dirigidos. También es posible desarrollar y compilar *drivers* para todas las versiones mediante el uso de librerías y funciones que son comunes a todas ellas.



## Actividades

16. Busque información sobre la programación para distintas versiones de Windows, sobre qué librerías y funciones permiten que un *driver* se pueda ejecutar en plataformas de distintas versiones.

Para que sea posible compilar en *Windows*, hay que realizar unas tareas previas:

En primer lugar, sería necesario crear una carpeta (llamada por ejemplo "misdrivers") donde se van a almacenar el código del *driver* (llamado para el ejemplo "eldriver.c") y dos archivos más. Estos archivos son: SOURCES y MAKEFILE.

El fichero SOURCES (sin extensión) contiene información sobre cuál es el código fuente y qué *driver* se va a construir. Además, se pueden añadir las rutas para encontrar los *headers* (INCLUDES) y las librerías (LIB) que va a utilizar el *driver*:

TARGETNAME = midriver	→ Nombre del <i>driver</i> que se va a generar
TARGETPTH = .	
TARGETTYPE = DRIVER	
SOURCES = eldriver.c	→ El fichero del código fuente del <i>driver</i>

El contenido del fichero de "Makefile.def" es el siguiente:

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Esta información va a ser usada por el entorno de desarrollo, como se verá más adelante.



#### Nota

Un archivo Makefile es un fichero de texto. Puede contener comentarios, variables y reglas para ser usadas por otros programas.

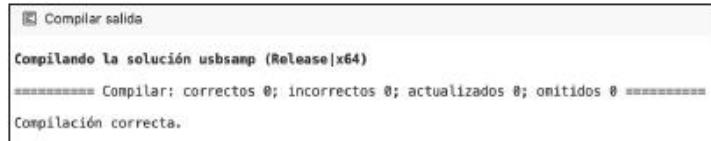


#### Actividades

17. Busque información sobre los *headers* y las librerías en C/C++. ¿Qué diferencia hay entre ellos? Escríbalas en una tabla.

Para compilar el *driver*, que se tiene previamente codificado, se abre el entorno de compilación del sistema operativo que se use. En el caso de que se use el sistema operativo *Windows*, esta opción se encuentra dentro de la herramienta *Visual Studio*.

```
Salida
Mostrar salida de: Compilación
Operación Compilar iniciada..
1----- Operación Compilar iniciada: Proyecto: usbsamp (Sys\Driver\usbsamp), configuración: Release x64 -----
2----- Operación Compilar iniciada: Proyecto: usbsamp (Exe\usbsamp), configuración: Release x64 -----
1>C:\Program Files\Microsoft Visual Studio\2022\Professional\MSBuild\Microsoft\VC\v170\Microsoft.CppBuild.targets(442,5):
1>Compilación del proyecto "usbsamp.vcxproj" terminada -- ERROR.
2>C:\Program Files\Microsoft Visual Studio\2022\Professional\MSBuild\Microsoft\VC\v170\Microsoft.CppBuild.targets(442,5):
2>Compilación del proyecto "usbsamp.vcxproj" terminada -- ERROR.
***** Compilación: 0 correcto, 2 erroneo, 0 actualizado, 0 omitido *****
***** 00:05,358 Transcurrido *****
```



#### Entorno para compilación de WDK de Windows

En la imagen se muestra la pantalla obtenida al compilar un *driver*.



Usando *Visual Studio*, se puede compilar el *driver* para distintas arquitecturas de los sistemas operativos.

También se puede llevar a cabo la comprobación del *driver* a través de la consola, sin olvidar que se debe usar la propia de la aplicación *Visual Studio* que se encuentra dentro del menú **Herramientas → Línea de comandos** y no en la consola propia del sistema operativo *Windows*.



#### Nota

Para moverse en la consola entre directorios, se utilizará “cd\” para subir al directorio principal. Para entrar en un directorio concreto, se utiliza “cd nombre de directorio”.

Dentro de la consola de *Visual Studio*, se puede ejecutar el comando **msbuild** para compilar el archivo. En el caso de que se produzcan errores, pueden visualizarse a través del modificador **-m** que indicará dónde se encuentran los errores.

```
Developer PowerShell Visual Studio Community 2022 17.4.33123.133
PS C:\ejemplos\sdminports> esbuild sdhc
MSBuild version 17.4.0-18d5ae85 for .NET Framework
Los proyectos de esta solución se van a compilar de uno en uno. Para habilitar la compilación en paralelo, agregue el modificador "-m".
Compilación iniciada a las 03/22/2022 10:07:11.
C:\Program Files (x86)\Windows Kits\10\build\10.0.22621.0\windows\driver\common\targets\235.5: error : 'win32' is not
a valid architecture for kernel mode drivers or UDR drivers [C:\ejemplos\sdminports\sdhc\inbox\sdhc.vcxproj]
Compilación del proyecto terminada "C:\ejemplos\sdminports\sdhc\inbox\sdhc.vcxproj" (destinos predeterminados) — ERRO
r.

Compilación del proyecto terminada "c:\ejemplos\sdminports\sdhc.sln" (destinos predeterminados) — ERROR.

Error al compilar.

'C:\ejemplos\sdminports\sdhc.sln' (destino predeterminado) (1) ->
'C:\ejemplos\sdminports\sdhc\inbox\sdhc.vcxproj' (destino predeterminado) (2) ->
\ValidateDriverProperties destino) ->
  C:\Program Files (x86)\Windows Kits\10\build\10.0.22621.0\windows\driver\common\targets\235.5: error : 'win32' is no
t a valid architecture for kernel mode drivers or UDR drivers [C:\ejemplos\sdminports\sdhc\inbox\sdhc.vcxproj]
  0 Advertencia(s)
  1 errores

Tiempo transcurrido 00:00:00.50
PS C:\ejemplos\sdminports> esbuild --sdhc
MSBuild version 17.4.0-18d5ae85 for .NET Framework
Compilación iniciada a las 03/22/2022 10:07:14.
1>El proyecto "C:\ejemplos\sdminports\sdhc.sln" en nodo 1 (destinos predeterminados).
1>\ValidateSolutionConfiguration
  Compilando la configuración de soluciones "Debug\Win32".
  >El proyecto "C:\ejemplos\sdminports\sdhc.sln" (1) está compilando "C:\ejemplos\sdminports\sdhc\inbox\sdhc.vcxproj" (2) en
  el nodo 1 (destinos predeterminados).
  >C:\Program Files (x86)\Windows Kits\10\build\10.0.22621.0\windows\driver\common\targets\235.5: error : 'win32' is not a valid a
rchitecture for kernel mode drivers or UDR drivers [C:\ejemplos\sdminports\sdhc\inbox\sdhc.vcxproj]
  2>Compilación del proyecto terminada "C:\ejemplos\sdminports\sdhc\inbox\sdhc.vcxproj" (destinos predeterminados) — ERRO
R.
  1>Compilación del proyecto terminada "C:\ejemplos\sdminports\sdhc.sln" (destinos predeterminados) — ERRO
R.

Error al compilar.

'C:\ejemplos\sdminports\sdhc.sln' (destino predeterminado) (1) ->
'C:\ejemplos\sdminports\sdhc\inbox\sdhc.vcxproj' (destino predeterminado) (2) ->
\ValidateDriverProperties destino) ->
  C:\Program Files (x86)\Windows Kits\10\build\10.0.22621.0\windows\driver\common\targets\235.5: error : 'win32' is not a valid
architecture for kernel mode drivers or UDR drivers [C:\ejemplos\sdminports\sdhc\inbox\sdhc.vcxproj]
  0 Advertencia(s)
  1 errores

Tiempo transcurrido 00:00:00.44
PS C:\ejemplos\sdminports>
```

Pantalla para realizar la compilación del driver en Windows

### Instalación del controlador

Una vez compilado, se procede a la carga o instalación del controlador. Para ello es necesario copiar el fichero que se ha generado en la compilación, con extensión **.sys**, en el directorio del sistema donde se almacenan los *drivers* (c:\Windows\System32\drivers).

Nombre	Fecha de modificación	Tipo	Tamaño
DriverData	05/06/2021 14:10	Carpeta de archivos	
en-US	05/06/2021 19:40	Carpeta de archivos	
es-ES	26/10/2022 8:55	Carpeta de archivos	
etc	25/04/2022 18:37	Carpeta de archivos	
UMDF	26/10/2022 8:55	Carpeta de archivos	
wd	11/11/2022 11:17	Carpeta de archivos	
3ware.sys	05/06/2021 14:04	Archivo de sistema	105 KB
1384ohci.sys	11/05/2022 8:47	Archivo de sistema	288 KB
acpisys	22/09/2022 11:42	Archivo de sistema	818 KB
AcpiDevsy	11/05/2022 8:47	Archivo de sistema	52 KB
acpiexsys	05/06/2021 14:04	Archivo de sistema	161 KB
acpipagrsy	11/05/2022 8:47	Archivo de sistema	44 KB
acipipltys	11/05/2022 8:47	Archivo de sistema	46 KB
acpitimesy	11/05/2022 8:47	Archivo de sistema	46 KB
Acx01000.sys	05/06/2021 14:04	Archivo de sistema	684 KB
adp80xx.sys	05/06/2021 14:04	Archivo de sistema	1.109 KB
afd.sys	16/06/2022 18:30	Archivo de sistema	666 KB
afunixsys	24/07/2022 16:45	Archivo de sistema	80 KB
agilevpn.sys	25/05/2022 16:22	Archivo de sistema	136 KB
ahcache.sys	05/06/2021 14:05	Archivo de sistema	332 KB
amdgpio2.sys	05/06/2021 14:04	Archivo de sistema	18 KB
amdi2c.sys	05/06/2021 14:04	Archivo de sistema	45 KB
amrik80xx	11/11/2022 9:59	Archivo de sistema	218 KB

463 elementos

Pantalla que muestra la carpeta donde Windows almacena los drivers, los archivos con extensión ".sys"

A partir de este momento, se debe registrar el controlador para que Windows lo trate como un servicio del sistema.



### Nota

Un servicio de Windows es una aplicación que se ejecuta en segundo plano, es decir, sin que haya interacción con el usuario.

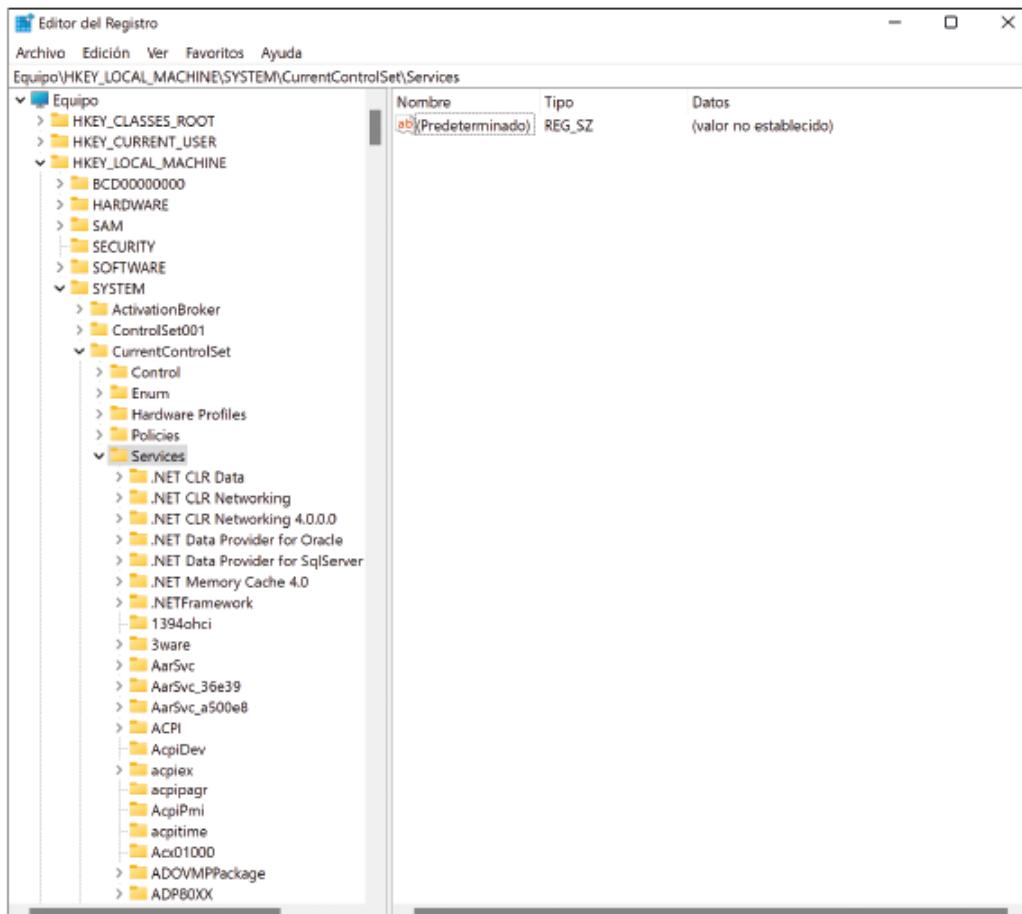
Los servicios pueden configurarse para que se inicialicen cuando se inicie la sesión de Windows. También pueden ser pausados o inicializados en cualquier momento.

El registro de un *driver* como un servicio del sistema puede hacerse de tres formas:

- Mediante la creación de un **cargador/eliminador** de servicios propio, utilizando funciones del Win32. La función que se utiliza para crear el servicio es **CreateService** y para eliminarlo **DeleteService**.  
Estas funciones lo que hacen básicamente es crear o eliminar unos valores en el **Registro de Windows**. El Registro es una base de datos jerárquica en la que se almacena la configuración del sistema operativo *Windows*.
- Creando de forma **manual** las entradas en el registro de Windows. Para añadir una nueva entrada en el registro de *Windows*, es necesario situarse dentro de la jerarquía de carpetas en:

HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Services\

En la imagen que se muestra a continuación se ve el Registro de *Windows* y dónde hay que posicionarse para la creación de la nueva clave.



*Pantalla del Registro de Windows donde están almacenados los datos de los servicios*

Una vez que situados en esta posición, se crea una carpeta con el nombre del nuevo *driver*, lo que se denomina una nueva **clave**. Además, se crearán una serie de valores, **subclaves**, que se van a utilizar para la carga del *driver*.

Valor del registro	Descripción	En el ejemplo
<b>DisplayName</b>	Nombre del nuevo <i>driver</i> que aparece en la lista de servicios.	eldriver
<b>ImagePath</b>	Directorio donde se ha copiado el fichero ".sys"; si se ha copiado en el directorio de Windows solo es necesario el nombre del <i>driver</i> .	eldriver
<b>Start</b>	Indica cómo se va a inicializar el <i>driver</i> , es un valor entero que significa lo siguiente: 0. Se inicializa con el cargador del <i>kernel</i> (Boot). 1. Se inicializa con el subsistema de E/S (System). 2. Se inicializa con el administrador de servicios. Indica que se inicializa automáticamente. 3. Esta configuración hace que el <i>driver</i> tenga que ser inicializado manualmente. Esta opción es recomendable para la fase de test. 4. Con este valor se indica que el <i>driver</i> está deshabilitado.	3 (para pruebas)
<b>Type</b>	Indica el tipo de servicio, existen ocho tipos pero en este caso el que interesa es modo <i>kernel</i> , el valor 1.	1

■ Una opción más recomendable, por ser más sencilla, es mediante el uso de herramientas que hagan la creación de claves en el Registro de Windows de forma directa.

Una de estas herramientas puede ser, por ejemplo, **OSR Driver Loader**. Esta herramienta funciona de forma muy simple, ya que solo es necesario seleccionar el nuevo *driver* y se encarga de forma automática de registrarlo como un servicio, además lo ejecuta. También permite parar el servicio y eliminarlo, si fuese necesario.



**Nota**

Un *driver* solo necesita ser registrado una vez.

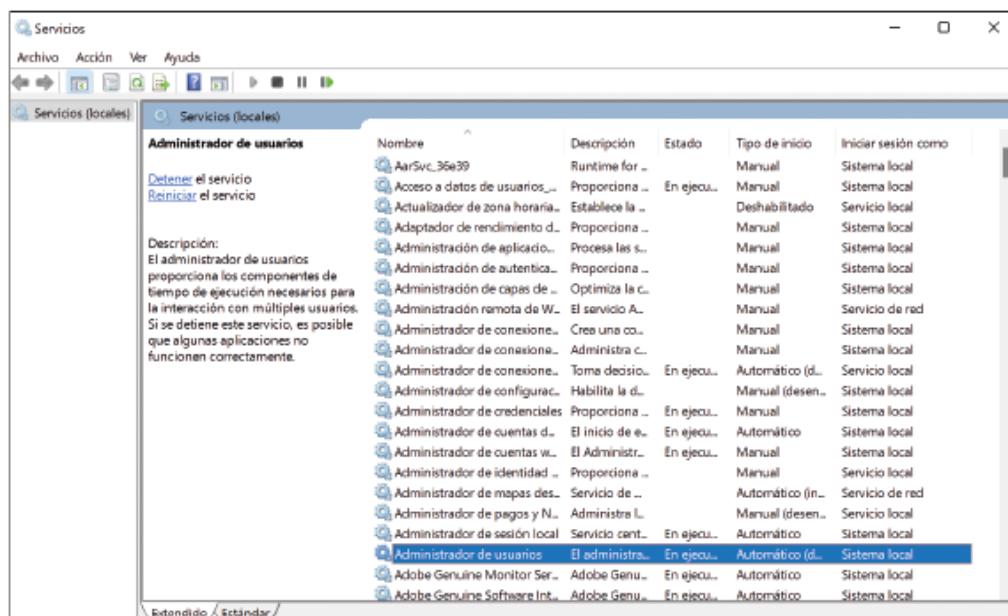


## Actividades

18. Busque información sobre las funciones para crear y eliminar servicios de Win32, CreateService y DeleteService, documentadas en las librerías MSDN de Microsoft.

Para comprobar que se ha registrado el *driver* correctamente y que está el servicio activo, se puede acceder a la pantalla de **Servicios** de Windows.

Además de obtener esta información, es posible detener desde aquí el servicio e inicializarlo de forma manual, como se ve en la imagen mostrada a continuación.



Pantalla de Servicios de Windows (aparece resaltado dónde parar y reiniciar el servicio señalado)



## Nota

El acceso a la pantalla de Servicios de Windows, se puede realizar de distintas maneras:

Versión	Acceso
Windows 10 y posteriores	Mediante el uso de la combinación de las teclas [Windows] + [R]
Desde Consola	Con el comando services.msc Para arrancar un servicio desde la consola se debe introducir la orden <i>net start</i> y el nombre del servicio que se quiere iniciar. Si se quiere parar el servicio la orden será <i>net stop servicio</i> . Para pausar un servicio la orden es <i>net pause servicio</i> . Para reanudarlo será <i>net continue servicio</i> .



### Actividades

19. Acceda a la pantalla de servicios del sistema y preste especial interés a los servicios que pertenezcan a *drivers* que controlen los dispositivos que hay conectados al sistema.

### Linux

Los *drivers* en *Linux* se empaquetan en **módulos** o se compilan directamente dentro del ***kernel***. Si se utiliza la primera opción, es decir, tratarlos de forma independiente, los módulos pueden ser cargados y descargados en tiempo de ejecución. De esta manera, el sistema es más flexible que si el módulo es compilado dentro del *kernel*.

Antes de comenzar a compilar un nuevo *driver*, una buena práctica a la hora de guardar y compilar el código fuente es crear un directorio dentro del “**/home**” del usuario, es decir, en “/home/miusuario/misdrivers”. Este será el directorio de trabajo.



### Nota

Cuando se utiliza un terminal en *Linux*, se puede observar que finaliza la línea de comandos con un símbolo \$ o #.

El símbolo \$ significa que se está en el sistema operativo como usuarios, mientras que el símbolo # avisa que se está como superusuarios.

### **Compilar en Linux**

Una vez codificado el nuevo *driver* y almacenado en la carpeta destinada para ello, denominada para el ejemplo "misdrivers", contendrá el fichero con el **código fuente** del *driver*, con extensión ".c", llamado para el ejemplo "eldriver.c".

Para compilar este fichero, es necesario acceder a la línea de comandos de *Linux* e introducir el siguiente comando:

```
$gcc -c /misdrivers/eldriver.c
```

O bien este mismo comando indicándole dónde debe mirar para buscar los ficheros "include":

```
$gcc -I/usr/src/linux/include -O -Wall -c /misdrivers/eldriver.c
```

Al finalizar la compilación, se genera un fichero ".o", en este ejemplo: "eldriver.o".



### **Actividades**

20. Buscar información sobre cómo compilar el *kernel* de *Linux*.

---

### **Carga del módulo**

Para que el *driver* que se ha creado ("eldriver.c") forme parte del *kernel* es necesario que se cargue.

En primer lugar, hay que entrar al sistema como un usuario *root* o bien acreditarse como este tipo de usuario.



### **Nota**

Un usuario *root* es una cuenta de usuario que posee todos los privilegios sobre el sistema (también se le llama superusuario), es decir, tiene un control absoluto de todo lo que ocurre en él.

Para acceder en modo root, o bien al iniciar el sistema se entra como este usuario o se cambia la identidad del usuario con el que se ha accedido al sistema, pudiendo utilizar para ello comandos como sudo o su.

Para realizar la carga, nuevamente desde la línea de comandos, se utiliza el comando **insmod**, que permite instalar un módulo dentro de un *kernel* en ejecución. Para el ejemplo que se está tratando se escribiría el siguiente comando:

```
# insmod /misdrivers/eldriver.o
```

O bien:

```
# insmod -f /misdrivers/midriver.o
```



#### Nota

La opción **-f** (**-force**) en el comando insmod permite cargar el módulo, aunque la versión del núcleo que se está ejecutando y la versión del núcleo en la que se compiló el fichero, que se toma como parámetro, sean diferentes.

Una vez terminada la carga, se puede **comprobar que se ha realizado de forma correcta** utilizando un comando que muestre en pantalla una lista con todos los módulos que están instalados. Dicho comando es:

```
# lsmod
```

#### Descarga del módulo

Para descargar módulos del *kernel* se utiliza el comando **rmmmod**, que descarga un conjunto de módulos con la restricción de que no estén siendo utilizados y que no se hagan referencias a ellos desde otros módulos.

En el ejemplo que se está tratando sería así:

```
# rmmod eldriver
```



## Actividades

21. Busque información sobre los parámetros que admite el comando insmod y para qué sirven.

## 7. Distribución de controladores de dispositivos

Una vez que se ha programado, compilado y testeado el nuevo *driver*, es posible que este entre a formar parte de las nuevas distribuciones de los sistemas operativos más usados, pero para ello debe cumplir una serie de requisitos.

En este apartado, se describen los pasos que hay que dar para que un *driver* propio pueda ser distribuido. Para ello, es necesario distinguir entre los distintos entornos de sistemas operativos.

### 7.1. Windows

Para que el *driver* que se ha programado pueda ser distribuido por *Windows*, se deben cumplimentar muchos pasos y certificaciones. Como cabe esperar, no es una tarea fácil. *Windows* debe estar seguro de que ese *driver* cumple con el objetivo para el que se diseñó, no va a producir errores, a priori, y es realmente necesario.

Para publicar el *driver* se deben observar los siguientes pasos:

#### Crear un paquete de distribución

En este paquete hay que incluir los archivos imprescindibles a la hora de instalar el *driver*. Evidentemente, todos estos archivos deben ser compatibles con *Windows*. Normalmente, un paquete de un *driver* incluye los siguientes componentes:

<b>Archivos propios del driver</b>	<b>El fichero con extensión “.sys”.</b>
Archivos de instalación	<p><b>Fichero de información (“.INF”)</b> con la configuración del dispositivo.</p> <p><b>Fichero con un catálogo del driver (“.Cat”)</b>, que contiene información encriptada que permite a <i>Windows</i> verificar que el paquete no se ha alterado una vez que se haya publicado.</p> <p>Para asegurarse de que no se ha alterado, debe ser firmado digitalmente.</p> <p><b>Otros</b></p> <p>Pueden contener ficheros que completen el funcionamiento del <i>driver</i>, por ejemplo una aplicación para instalar el dispositivo o un ícono, etc.</p>



### Nota

Las firmas digitales se basan en la tecnología de clave pública de Microsoft, que consiste en combinar **Microsoft Authenticode** con los servicios de una entidad emisora de certificados de confianza.



### Actividades

22. Busque información sobre la tecnología de encriptación de datos basada en clave privada y clave pública.

### Firmar el driver

Todos los *drivers* que se ejecuten sobre las versiones de 64-bits de *Windows* están obligados a ser firmados antes de que *Windows* pueda cargarlos, pero no se requiere para los que se ejecuten sobre 32-bits.

Para firmar un *driver* se necesita un **certificado**. Se puede utilizar un certificado propio durante la fase de desarrollo y pruebas, pero para poder publicarlo es necesario que el certificado sea realizado por una entidad de confianza (por ejemplo *VeriSign Certificate*).



### Actividades

23. Busque información sobre entidades de confianza, para *Microsoft*, que permitan emitir certificados de forma gratuita.

### Implementar el *driver* en un ordenador de prueba

La herramienta WDK proporciona unas funcionalidades que permiten **testear** el *driver* en un **equipo de prueba**, de manera que se utilizará un equipo en el que se va a desarrollar, compilar y depurar el *driver*, el llamado **host**, y otro en el que se va a instalar y probar el *driver*.

También es posible conectar el equipo *host* a varios equipos de prueba con intención de probar distintos escenarios.

#### Certificar el *driver*

Para que el nuevo *driver* esté disponible para *Windows*, es necesario certificarlo. La herramienta que se utiliza para esto es lo que se denomina *Windows Hardware Lab Kit* (HLK), o Kit de Laboratorio de *Hardware* de *Windows*.

#### Distribuir el paquete *driver*

La distribución de paquetes se hace de forma segura con el programa de Microsoft **Windows Update**. Para que este paquete de *driver* sea publicado para su distribución, es necesario que previamente haya pasado la certificación con el Kit de Laboratorio de *Hardware* de *Windows* y haya sido firmado digitalmente con el **WHQL**.



#### Nota

Para obtener una firma que permita distribuir un paquete de *drivers* con WHQL, se utiliza una parte del Kit de Laboratorio de *Hardware* de *Windows*.

Esta firma consiste en un fichero de tipo catálogo firmado digitalmente. La firma no cambia los archivos binarios ni los de tipo INF que estén incluidos en el paquete de *driver* firmado.

*Windows Update* impone además que se cumplan los siguientes requisitos adicionales:

- Confirmar que el paquete del *driver* es adecuado para el dispositivo para el que se ha creado.
- Que pueda ser legalmente distribuido.
- Que pueda ser descargado de forma automática.

23. Busque información sobre entidades de confianza, para *Microsoft*, que permitan emitir certificados de forma gratuita.

### Implementar el *driver* en un ordenador de prueba

La herramienta WDK proporciona unas funcionalidades que permiten **testear** el *driver* en un **equipo de prueba**, de manera que se utilizará un equipo en el que se va a desarrollar, compilar y depurar el *driver*, el llamado **host**, y otro en el que se va a instalar y probar el *driver*.

También es posible conectar el equipo *host* a varios equipos de prueba con intención de probar distintos escenarios.

### Certificar el *driver*

Para que el nuevo *driver* esté disponible para *Windows*, es necesario certificarlo. La herramienta que se utiliza para esto es lo que se denomina *Windows Hardware Lab Kit* (HLK), o Kit de Laboratorio de *Hardware* de *Windows*.

### Distribuir el paquete *driver*

La distribución de paquetes se hace de forma segura con el programa de Microsoft **Windows Update**. Para que este paquete de *driver* sea publicado para su distribución, es necesario que previamente haya pasado la certificación con el Kit de Laboratorio de *Hardware* de *Windows* y haya sido firmado digitalmente con el **WHQL**.



#### Nota

Para obtener una firma que permita distribuir un paquete de *drivers* con WHQL, se utiliza una parte del Kit de Laboratorio de *Hardware* de *Windows*.

Esta firma consiste en un fichero de tipo catálogo firmado digitalmente. La firma no cambia los archivos binarios ni los de tipo INF que estén incluidos en el paquete de *driver* firmado.

*Windows Update* impone además que se cumplan los siguientes requisitos adicionales:

- Confirmar que el paquete del *driver* es adecuado para el dispositivo para el que se ha creado.
- Que pueda ser legalmente distribuido.
- Que pueda ser descargado de forma automática.



## Actividades

24. Debido a que los requisitos impuestos por *Windows Update* se modifican con frecuencia, compruebe qué requisitos están disponibles en este momento en la web de *Windows Update* para la publicación de drivers.

### 7.2. Linux

El núcleo de *Linux* está en constante desarrollo, sujeto a cambios continuos y abierto a aportaciones de cualquier desarrollador.

A pesar de esto, cuenta con una infraestructura que permite que siempre haya una última versión estable.

Los desarrolladores del *kernel* de *Linux* deben compilar sus aportaciones partiendo de esta última versión estable.

En el código fuente que se pretende incluir en el *kernel*, es necesario añadir, a modo de comentario, el número de la versión del núcleo en el que se compilaron los ficheros de cabecera.

El *kernel* de *Linux* cuenta con varios tipos de categorías de versiones activas que se van a describir a continuación.

#### Prepatch

Este tipo de *kernel* es pre-distribución y está orientado a desarrolladores. Debe ser compilado desde las fuentes del *kernel*, ya que normalmente contiene nuevas características que deben ser testeadas antes de que lleguen a una versión estable del núcleo.

Estas versiones de *kernel* están mantenidas por Linus Torvalds.

#### Principal (*mainline*)

El árbol principal (o *mainline*) es el árbol donde todas las nuevas características se introducen, además de los nuevos desarrollos. Es también mantenida por Linus Torvalds. Y pasan a estar disponibles cada dos o tres meses.

## Estable

Una vez que se liberan las versiones principal (*mainline*), este *kernel* pasa a ser considerado estable. Ya que ha incluido los nuevos desarrollos y parches después de haber sido testeado por los responsables de mantener la integridad del *kernel*.

## A largo plazo (*longterm*)

Normalmente, se producen varios mantenimientos a largo plazo para proporcionar una puerta deatrás a los errores que se detectan en versiones antiguas. Aunque solo se aplican si son errores importantes.



### Sabía que...

Linus Torvalds, ingeniero de software nacido en Finlandia en 1969, es el creador del sistema *Linux*. Lo desarrolló a partir de un sistema operativo libre llamado *Minix*. Actualmente, mantiene el desarrollo del *kernel* de *Linux*.



### Actividades

25. Busque un comando que se ejecute en el terminal de *Linux* para conocer qué distribución se está ejecutando.
26. ¿Dónde se puede ver cuál es la última versión estable para el *kernel* de *Linux*?
27. ¿A qué se denomina núcleo vainilla? Busque información.

Según la Linux Kernel Organization, para que se pueda distribuir un nuevo *driver* en las futuras versiones de *kernel* que se vayan a liberar debe hacerse lo siguiente:

- Conseguir que un grupo de personas que tengan el *hardware* al que va dirigido el *driver* **prueben** si cumple con el nuevo desarrollo las expectativas para las que se diseñó. Una forma de reunir a este grupo de personas es escribiendo en foros o en grupos de noticias para conseguir colaboración. Si no se comprueba que ha sido testeado por varias personas, no podrá incluirse en las nuevas versiones del *kernel*.
- Debido al tiempo que esto lleva, es posible que se haya liberado la siguiente versión del *kernel*, con lo cual, si es necesario, hay que desarrollar un parche que permita **adaptarlo** a la nueva versión.
- Por último, se pone el *driver* en un **servidor FTP** y se escribe en la **lista de correos** de la organización una entrada en la que aparezca la **URL** para acceder al *driver*. Además, se acompaña la entrada de una descripción de lo que hace el

*driver* con el dispositivo a tratar. En esta lista de correos, la petición le llegará a la persona encargada de revisar esta parte del *kernel* y, si lo consideran una aportación interesante, lo incluirán en la nueva versión.



## Definición

### Servidor FTP

Programa que se ejecuta en un equipo conectado a internet y que permite el intercambio de datos. Un ejemplo de este tipo de programas gratuito es Filezilla.

### Lista de correos

Función especial del correo electrónico que permite enviar un mensaje de forma simultánea a un grupo de personas.

### URL

Cadena de caracteres que indica una dirección única a través de Internet que permite el acceso a un recurso.



## Nota

Si se pretende distribuir *drivers* para plataformas de 32-bits y de 64-bits, es necesario preparar un paquete de instalación del *driver* independiente para cada una de ellas.

## 8. Particularidades en el desarrollo de dispositivos en sistemas operativos de uso común

Cada sistema operativo tiene su forma de comunicarse con el *hardware* del equipo. A la hora de desarrollar un controlador de dispositivo o *driver* es necesario conocer cuáles son las particularidades de cada plataforma, ya que el diseño de este *software* va a venir impuesto por ellas.

En este apartado se van a definir las particularidades que distinguen un sistema operativo y otro.

## 8.1. Sistemas Windows

Para los sistemas *Windows*, la definición de *driver* es más extensa que solo considerarlo como un componente de *software* que permite que el sistema operativo y un dispositivo se comuniquen entre sí.

En esta plataforma, no todos los *drivers* se comunican directamente con un dispositivo, ya que con frecuencia se tienen varias capas de *drivers* dispuestos en una **pila**.

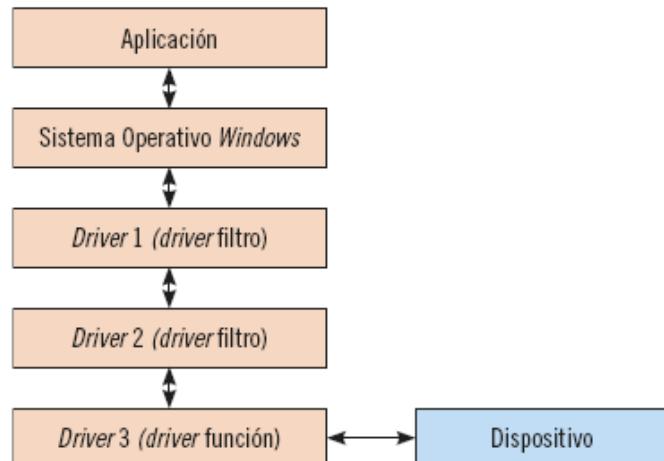
Por ello, se puede observar que el que se encuentra más arriba de la pila es el que interactúa directamente con la aplicación del usuario.

El que está en el nivel más bajo de la pila actúa directamente con el dispositivo.

Los componentes que se encuentren en las capas intermedias permiten que se comuniquen unas capas con otras.

Un ejemplo de esta comunicación se produce cuando es necesario cambiar de formato la información, para que sea entendible por la capa más cercana al *hardware* y la capa más cercana a las aplicaciones de usuario.

En la imagen que se muestra a continuación, puede verse esta pila de *drivers*.



De manera que para el sistema operativo *Windows* un *driver* es: "Cualquier componente *software* que observa o participa en la comunicación entre el sistema operativo y un dispositivo".

Un ejemplo de *driver* que no está asociado a ningún dispositivo puede ser aquel que acceda a las estructuras de datos del sistema operativo. Este tipo de accesos solo puede hacerse en modo *kernel*.

Para desarrollar esta herramienta, se crearían, por lo tanto, dos componentes:

- Un componente que se ejecute en modo de usuario y que interactúe mediante una interfaz de usuario.
- Un componente que se ejecute en modo núcleo y que tenga acceso a los datos del sistema operativo.

El componente que se ejecuta en el modo de usuario es una **aplicación**, y el componente que se ejecuta en modo *kernel* es un **controlador de software o driver**.



### Importante

Tanto *Windows* como *Linux*, tienen disponibles las versiones de sus sistemas operativos para ser instaladas en los servidores. Son fácilmente identificables estas versiones al incorporar la palabra “*Server*” en sus denominaciones comerciales.

Estos sistemas operativos tienen un funcionamiento similar al resto, permitiendo configurar permisos específicos para cada uno de los usuarios del equipo.

Actualmente *Microsoft* ofrece su sistema operativo “*en la nube*” denominado *Windows 365*, de forma que se pueda ejecutar en cualquier dispositivo.

---

## Aspectos básicos del hardware

Todos los elementos que integran el *hardware* constituyen la arquitectura interna del ordenador o dispositivo. A continuación se describen dichos elementos.

### Registros de dispositivo

Los *drivers* se comunican con un dispositivo leyendo y escribiendo en los registros asociados al dispositivo:

- Command: controlan el dispositivo.
- Status: muestra el estado actual del dispositivo.
- Data: se emplean para el paso de datos.

Puerto, es una dirección para lectura y escritura

Existen unas macros que permiten acceder a estos puertos:

Función	Descripción
READ_PORT_XXX	Lee un valor de un puerto E/S
WRITE_PORT_XXX	Escribe un valor en un puerto E/S
READ_REGISTER_XXX	Lee un valor de un registro E/S
WRITE_REGISTER_XXX	Escribe un valor en un registro E/S

### Interrupciones de dispositivo

try catch

Las interrupciones permiten que el dispositivo llame la atención de la CPU. De esta forma se mejora el rendimiento del sistema.

### Mecanismos de transferencia de datos

Las técnicas que se usan para el paso de datos de la CPU a la memoria son:

- E/S programada.
- DMA.
- Búfer compartido.

### Autoreconocimiento y autoconfiguración del dispositivo

Gracias a las nuevas arquitecturas de bus, se incorpora una tecnología con la cual un dispositivo se autoreconoce.

De esta forma, muestra su presencia al sistema y se autoconfigura de forma *software*, asegurándose una asignación de recursos.

Un dispositivo debe, junto con el bus, generar una señal que notifique que se ha insertado o retirado.

Al insertarse, debe identificarse y dar una lista de los recursos que necesita. Debe enviar:

- El identificador del fabricante.
- El tipo de dispositivo.
- Requisitos de E/S.
- Las interrupciones que requiere.
- Requisitos de DMA.
- Requisitos de memoria de dispositivo.

### Procesando de E/S en modo *kernel*

La utilización de este modo de procesamiento, tiene el inconveniente de que se utilizan elementos comunes necesarios para que funcione el sistema operativo con el riesgo que conlleva funcionar.

#### Contexto de ejecución en modo *kernel*

El contexto describe el estado en el que se encuentra un sistema cuando se ejecuta una instrucción. Para almacenar este estado, se almacena el contenido de todos los registros de la CPU, etc.

En Windows Server se recogen tres contextos de ejecución:

Tipo de contexto	Descripción
Para excepciones o traps	Es cuando ha ocurrido una excepción
Para interrupciones	Cuando se produce una interrupción
Para hilos de modo <i>kernel</i>	Cuando parte de un código se ejecuta en un hilo en modo <i>kernel</i>

#### Llamadas a procedimientos diferidos (DPC)

thread modo *kernel*

Es un mecanismo de Windows que permite que tareas de alta prioridad, por ejemplo una interrupción, pasen a tener una menor prioridad y permitan la ejecución de otros códigos, para evitar penalizar el rendimiento del sistema.

#### Acceso a búferes de usuario

Para evitar que desde procesos en modo *kernel* se modifiquen los búferes de datos de usuario, el Administrador de E/S permite el acceso a estos búferes de dos formas:

Acceso	Descripción
<b>E/S mediante búfer</b>	El administrador E/S copia el búfer en la memoria RAM, de manera que el dispositivo puede hacer uso de esta copia. Una vez terminado el dispositivo con la operación, se copia de nuevo al búfer de usuario
<b>E/S directa (DMA)</b>	El administrador bloquea los datos para que no se modifiquen

### Rutinas de inicialización y limpieza de un driver

Cuando un *driver* se carga o descarga del sistema, se deben llevar a cabo varias acciones:

Rutina	Descripción
<b>DriverEntry</b>	<p>Esta rutina es ejecutada por el <i>driver</i> cuando se carga en el sistema. Esta carga debe poder realizarse de forma dinámica, ya que puede darse en cualquier momento.</p> <p>Dentro de esta rutina, se realizan tareas del estilo:</p> <ul style="list-style-type: none"> <li>- Localizar el <i>hardware</i> a controlar.</li> <li>- Reservar los recursos que necesita.</li> <li>- Dar un nombre al dispositivo en el sistema para que pueda accederse a él.</li> </ul>
<b>Unload</b>	Se llama cuando el <i>driver</i> es eliminado. Tiene que deshacer las rutinas que se hayan hecho en DriverEntry.
<b>Reinitialize</b>	Se utiliza cuando el <i>driver</i> tiene que inicializarse a sí mismo.
<b>AddDevice</b>	Para los dispositivos PnP en lugar de utilizar DriverEntry se utiliza AddDevice, que se encarga de las mismas funciones.

### Rutinas para peticiones E/S

Cuando el administrador de E/S recibe una petición que procede de una aplicación en modo usuario se llama a estas rutinas.

Rutinas	Descripción
<b>Operaciones Open</b>	El <i>driver</i> debe contar con una rutina CreateDispatch que permite manejar la petición Win32 CreateFile.
<b>Operaciones Close</b>	El <i>driver</i> debe contar con una rutina CloseDispatch que permite manejar la petición Win32 CloseHandle.
<b>Operaciones de dispositivo</b>	Dependiendo del dispositivo el <i>driver</i> puede tratar transferencias de datos y control de operaciones. Son ReadFile, WriteFile y DeviceIoControl, para permitir al usuario leer y escribir datos o configurar el dispositivo.

### **Rutinas para transferencia de datos**

Los grupos de rutinas que se emplean para tratar las transferencias de datos son:

Rutinas	Descripción
<b>Start I/O</b>	Se produce cuando un dispositivo comienza la transferencia de datos.
<b>Tratamiento de interrupción (ISR)</b>	Cada vez que un dispositivo genera una interrupción se utilizan estas rutinas.
<b>DPC</b>	Se utilizan cuando sea necesario utilizar las DPC.

### **Sincronización de recursos**

Rutinas que se utilizan para sincronizar el acceso a un recurso, ya que si se ejecuta un código en modo *kernel* no se permite bloquear un hilo en espera de un recurso.

Existen tres tipos de rutinas de sincronización:

Rutinas	Descripción
<b>ControllerControl</b>	Comienza una operación de transferencia de datos.
<b>AdapterControl</b>	Comienza una operación de transferencia de datos con DMA.
<b>SynchCriticalSection</b>	Sirve para acceder a los recursos del <i>hardware</i> o datos del <i>driver</i> que están compartidos con una rutina de interrupción (InterruptService) del <i>driver</i> .

### **Objetos en modo kernel**

La utilización de objetos en modo *kernel* permite su uso sin alterar los elementos que necesita el sistema operativo para funcionar.

### I/O Request Packets (IRP)

Las transacciones E/S en Windows se realizan a través de unas estructuras denominadas IRP.

Estas estructuras de datos son parcialmente opacas y representan un paquete de petición E/S.

Cada operación E/S de usuario es gestionada, a través del administrador E/S, creando una IRP en la memoria.

Según el manejador del dispositivo y la operación que haya solicitado el usuario, se pasa la IRP a la rutina de usuario apropiada.

Una vez terminada la rutina, el *driver* indica en el IRP un código de estado que será utilizado por el usuario que realizó la operación.



#### Nota

Un IRP es la estructura básica del administrador de E/S que se usa para comunicarse con los *drivers* y que permite que los *drivers* se comuniquen entre sí.

Este tipo de paquetes consiste en dos partes:

- Cabecera o parte fija del paquete: aquí el administrador E/S almacena información sobre la petición original, por ejemplo la dirección del dispositivo, etc. También lo utilizan los *drivers* para almacenar el estado final de la petición.
- Pila E/S: en la pila hay una posición por cada *driver* de la pila de *drivers* asociados a la petición. Cada posición de la pila contiene parámetros, código de funciones y los contextos que usa el driver correspondiente para hacer lo que se supone que se espera que haga con el dispositivo.

Un ejemplo de cómo se definiría una estructura del tipo IRP sería el siguiente:

```
typedef struct _IRP  
{
```

```
    SHORT Type;
    WORD Size;
    PMDL MdlAddress;
    ULONG Flags;
    ULONG AssociatedIrp;
    LIST_ENTRY ThreadListEntry;
    IO_STATUS_BLOCK IoStatus;
    CHAR RequestorMode;
    UCHAR PendingReturned;
    CHAR StackCount;
    CHAR CurrentLocation;
    UCHAR Cancel;
    UCHAR CancelIrql;
    CHAR ApcEnvironment;
    UCHAR AllocationFlags;
    PIO_STATUS_BLOCK UserIosb;
    PKVENT UserEvent;
    UINT64 Overlay;
    PVOID CancelRoutine;
    PVOID UserBuffer;
    ULONG Tail;
} IRP, *PIRP;
```

### **Objeto del controlador**

En un *driver* en *Windows* siempre debe aparecer la rutina **DriverEntry**, con el mismo nombre. Cuando el administrador E/S necesita localizar otras funciones busca en esta y es aquí donde encuentra un catálogo que le permite alcanzar los códigos de las distintas funciones.

### **Objetos del dispositivo**

Estos objetos mantienen información sobre las características y el estado que tiene el dispositivo en un momento determinado. Además, permiten al administrador E/S y al *driver* manejar dicho dispositivo.

Estos objetos se crean en la rutina **DriverEntry** y se destruyen en la rutina **Unload**.

Un ejemplo de la declaración de la función DriverEntry es:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject,PUNICODE_STRING pRegistryPath);
```

El primer parámetro que aparece es **DRIVER\_OBJECT**. Esta estructura de datos representa al *driver*.

Este objeto va a contener un puntero a **DEVICE\_OBJECT** (objetos del dispositivo), que consiste en una estructura que representa a un dispositivo en particular.

Un *driver* puede manejar varios dispositivos y, por lo tanto, el objeto del dispositivo es en realidad una lista a todos los dispositivos a los que este *driver* puede manejar.

El segundo parámetro, **RegistryPath**, especifica el camino para acceder a la clave de registro del *driver* (Regedit). En esta clave se almacena información específica del *driver*.



### Actividades

28. Busque información sobre qué es lo que devuelve la rutina DriverEntry. Si no contiene errores es STATUS\_SUCCESS, pero ¿qué devuelve si los tiene? [el código del error](#)
29. Busque información sobre cómo implementar en C/C++ una lista para poder desarrollar una función DriverEntry que contenga un DEVICE\_OBJECT que apunte a una lista de dispositivos.

[https://learn.microsoft.com/es-es/windows-hardware/drivers/ddi/wdm/ns-wdm-\\_device\\_object](https://learn.microsoft.com/es-es/windows-hardware/drivers/ddi/wdm/ns-wdm-_device_object)



### Aplicación práctica

**Es necesario realizar un *driver* en el sistema operativo Windows que permita saber cuándo se ha cargado un *driver*. Para ello, se necesita que aparezca un mensaje que indique que se ha producido la carga. ¿Cómo y dónde habría que programar esto?**

#### SOLUCIÓN

En todos los *drivers* en el sistema Windows la primera función que se ejecuta al cargar el *driver* es DriverEntry. Por lo tanto, es ahí donde se tiene que realizar la programación de este aviso.

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject,PUNICODE_STRING pRegistryPath)
{
    DbgPrint("El nuevo driver... se está cargando...\n");
    return STATUS_SUCCESS;
}
```

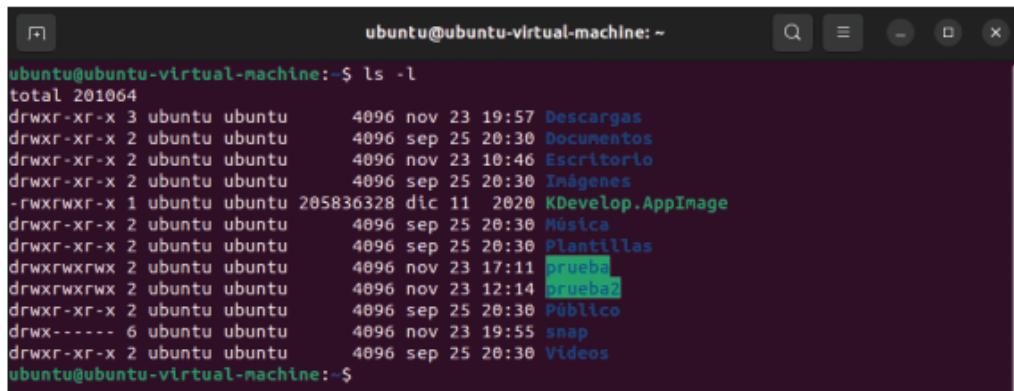
## 8.2. Sistemas Unix

Los *drivers* son una clase de módulo del *kernel* que proporciona una funcionalidad para un *hardware* específico.

En *Unix*, cada unidad *hardware* está representado en un fichero situado en el directorio **"/dev"** y que se llama archivos de dispositivos.

Los *drivers* realizan la comunicación que se establece con el dispositivo en nombre de un programa de usuario.

Un archivo de dispositivo tendría el siguiente formato (en el terminal, se hace: ls –l, y se muestra un listado con formato largo):



```
ubuntu@ubuntu-virtual-machine:~$ ls -l
total 201064
drwxr-xr-x 3 ubuntu ubuntu 4096 nov 23 19:57 Descargas
drwxr-xr-x 2 ubuntu ubuntu 4096 sep 25 20:30 Documentos
drwxr-xr-x 2 ubuntu ubuntu 4096 nov 23 10:46 Escritorio
drwxr-xr-x 2 ubuntu ubuntu 4096 sep 25 20:30 Imágenes
-rwxrwxr-x 1 ubuntu ubuntu 205836328 dic 11 2020 KDevelop.AppImage
drwxr-xr-x 2 ubuntu ubuntu 4096 sep 25 20:30 Música
drwxr-xr-x 2 ubuntu ubuntu 4096 sep 25 20:30 Plantillas
drwxrwxrwx 2 ubuntu ubuntu 4096 nov 23 17:11 prueba
drwxrwxrwx 2 ubuntu ubuntu 4096 nov 23 12:14 prueba2
drwxr-xr-x 2 ubuntu ubuntu 4096 sep 25 20:30 PÚblico
drwx----- 6 ubuntu ubuntu 4096 nov 23 19:55 snap
drwxr-xr-x 2 ubuntu ubuntu 4096 sep 25 20:30 Videos
ubuntu@ubuntu-virtual-machine:~$
```

En el ejemplo de arriba aparece un archivo de dispositivo que representa una partición de un disco duro (sda1).

Se observa que aparece un primer número y a continuación una coma. Este primer número se llama el **mayor número** del dispositivo. En este ejemplo es el 8.

El número que aparece detrás de la coma se denomina **menor número**. En el ejemplo es el 1.

El mayor número indica qué *driver* está siendo usado para acceder al dispositivo.

Cada *driver* solo tiene asignado un único mayor número, de manera que todos aquellos ficheros de dispositivo con este mayor número son controlados por el mismo *driver*.

El menor número le sirve al *driver* para distinguir entre los distintos dispositivos que controla.

La primera letra que aparece es "b". indica que se trata de un dispositivo de **bloque**. Si se tratase de un dispositivo de tipo **carácter**, aparecería una "c".

### Módulos kernel

Los módulos del *kernel* de *Linux* deben cumplir, como mínimo, lo siguiente:

- **Función de inicialización:** esta función se llama init\_module() y es invocada cuando se carga el módulo en el *kernel*.
- **Función final:** llamada cleanup\_module(), que es llamada justo antes de que el módulo sea quitado del *kernel*. Esta función deshace lo que se inicializó en la función init\_module().
- **Librería Linux/module.h:** ya que para realizar la depuración del código es prioritario el uso de la función printk(), es necesario incluir esta librería para indicar el nivel de prioridad que interesa mostrar en el mensaje, es decir, los KERN\_ALERT.



#### Nota

A partir de la versión de *Linux* 2.4 es posible cambiar el nombre de las funciones init\_module() y clean\_module(). Esto se hace con las macros module\_init() y module\_exit(). Estas macros se definen en <linux/init.h>. Lo único que hay que tener en cuenta es que se deben definir antes de llamar a las macros; si no, se producirá un error de compilación.

### Funciones disponibles para módulos

Los módulos del *kernel* de *Linux* son ficheros de objetos que se resuelven en el tiempo de carga. Las definiciones de los símbolos que contienen provienen del *kernel* en sí.

Las únicas funciones externas que se pueden utilizar, por lo tanto, son las proporcionadas por el *kernel*.

Hay que tener en cuenta la diferencia entre:

- **Funciones de biblioteca:** funciones de alto nivel que se ejecutan por completo en el espacio del usuario. Proporcionan una interfaz más cómoda, ya que en realidad estas funciones están compuestas de llamadas al sistema.
- **Llamadas al sistema (*system calls*):** se ejecutan en modo *kernel* en nombre del usuario. Son proporcionadas por el propio núcleo.



### Nota

Es posible programar los módulos del *kernel* para reemplazar las llamadas al sistema.

De hecho, eso es lo que hacen algunos *crackers*, modificar de forma maliciosa las llamadas al sistema y de esta forma se crean puertas deatrás al sistema o troyanos.



### Aplicación práctica

**Se necesita obtener un listado para comprobar qué identificador de driver y qué identificador individual tienen los distintos dispositivos de disco SCSI que contiene un sistema Linux concreto.**

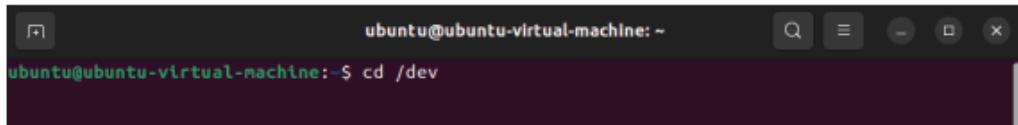
**Como referencia, cabe destacar que todos estos dispositivos tienen el prefijo "sda".**

**Por lo tanto, se pretende obtener un listado con formato largo por pantalla, que muestre todos los dispositivos de disco SCSI y los identificadores requeridos.**

### SOLUCIÓN

Para acceder a los dispositivos del sistema en *Linux*, hay que situarse, a través de la consola, en el directorio "/dev". Esto se hace de la siguiente manera:

Utilizando el comando cd /dev.

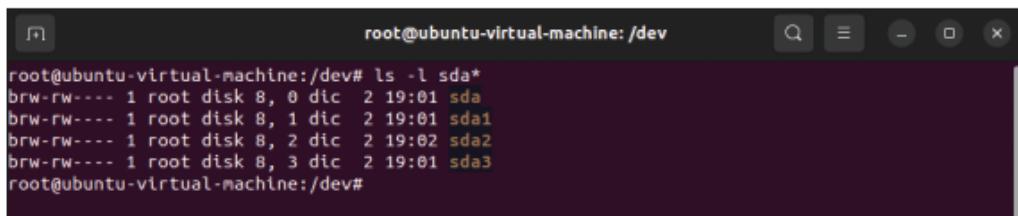


```
ubuntu@ubuntu-virtual-machine:~$ cd /dev
```

Pantalla de la consola del sistema operativo Ubuntu, con el comando `cd`

Una vez situados en el directorio donde se almacenan los archivos de dispositivos, se hace un listado filtrando por los dispositivos que empiecen por "sda", de la siguiente manera:

- Para realizar el listado se utiliza el comando `ls` seguido de la opción `-l`, con esto se indica que se muestre el formato largo.
- Por último, se añade el filtro, en este caso "`sda*`", el "\*" final indica que se quiere ver todos los ficheros que comiencen por sda.
- En la imagen que se muestra a continuación, aparece el comando utilizado y el resultado que se obtiene en la pantalla.



```
root@ubuntu-virtual-machine:/dev# ls -l sda*
brw-rw---- 1 root disk 8, 0 dic 2 19:01 sda
brw-rw---- 1 root disk 8, 1 dic 2 19:01 sda1
brw-rw---- 1 root disk 8, 2 dic 2 19:02 sda2
brw-rw---- 1 root disk 8, 3 dic 2 19:01 sda3
root@ubuntu-virtual-machine:/dev#
```

Pantalla de la consola del sistema operativo Ubuntu, con el comando `ls -l`

### El sistema de ficheros "/proc"

En Linux existe un mecanismo para que el *kernel* y los módulos del *kernel* puedan enviar información a los procesos. Este es el sistema de ficheros "/proc".

Es muy usado por cualquier parte del *kernel* que tenga que informar de algo. Por ejemplo: /proc/modules proporciona una lista con los módulos, /proc/meminfo contiene estadísticas de uso de memoria, etc.

### Hablando con los ficheros de dispositivos

Estos ficheros representan a los dispositivos físicos. La mayoría de los dispositivos se usan tanto para entradas como para salidas. Por lo tanto, tiene que haber un mecanismo para que los *drivers* para puedan leer y escribir del dispositivo.

Existe una función especial en *Unix*, llamada **ioctl**, que permite a las aplicaciones comunicarse o controlar un *driver* de un dispositivo.

A la función ioctl se le invoca con tres parámetros: el descriptor del archivo del dispositivo, un número ioctl y un parámetro de tipo entero para que se pueda usar por el programador para pasar cualquier cosa.



### Sabía que...

Existe una organización a gran escala que permite ayudar, a través de una comunidad de apoyo en la que está incluido el propio Linus Torvalds, al desarrollo y soporte del *kernel* de *Linux*. Esta comunidad promueve el intercambio libre y abierto de ideas. Además, ofrece los servicios esenciales y la infraestructura para que exista una colaboración continua que permita el progreso y la protección del *kernel* de *Linux*. Esta organización es la *Linux Foundation*. A esta comunidad puede unirse cualquiera que sienta pasión por el desarrollo del *kernel* y que quiera ofrecerla de forma libre.



### Actividades

30. Busque información sobre el grupo de trabajo CGL de la *Linux Foundation*.

## 8.3. Modos de instalación de controladores de dispositivo en sistemas operativos de uso común. Dispositivos Plug & Play

La instalación de los controladores de dispositivos debe estar aislada del conocimiento del *hardware* sobre el que se ejecuten. No es necesario conocer que el controlador va a estar mapeado en la dirección de memoria X o que para acceder a este controlador debe utilizarse la interrupción fija Y.

El usuario que vaya a instalar un *driver* concreto, no tiene por qué estar obligado a configurar *jumpers*, interruptores o cualquier otro elemento de este tipo. Por ejemplo, cualquier usuario puede utilizar un teclado y no necesitar conocer cómo es su circuitería ni cómo se comunica con el sistema operativo.

Con la tecnología *Plug and Play* (PnP) es posible instalar y desinstalar los dispositivos en el sistema sin tener conocimientos de cómo funciona el *hardware* del dispositivo en cuestión.

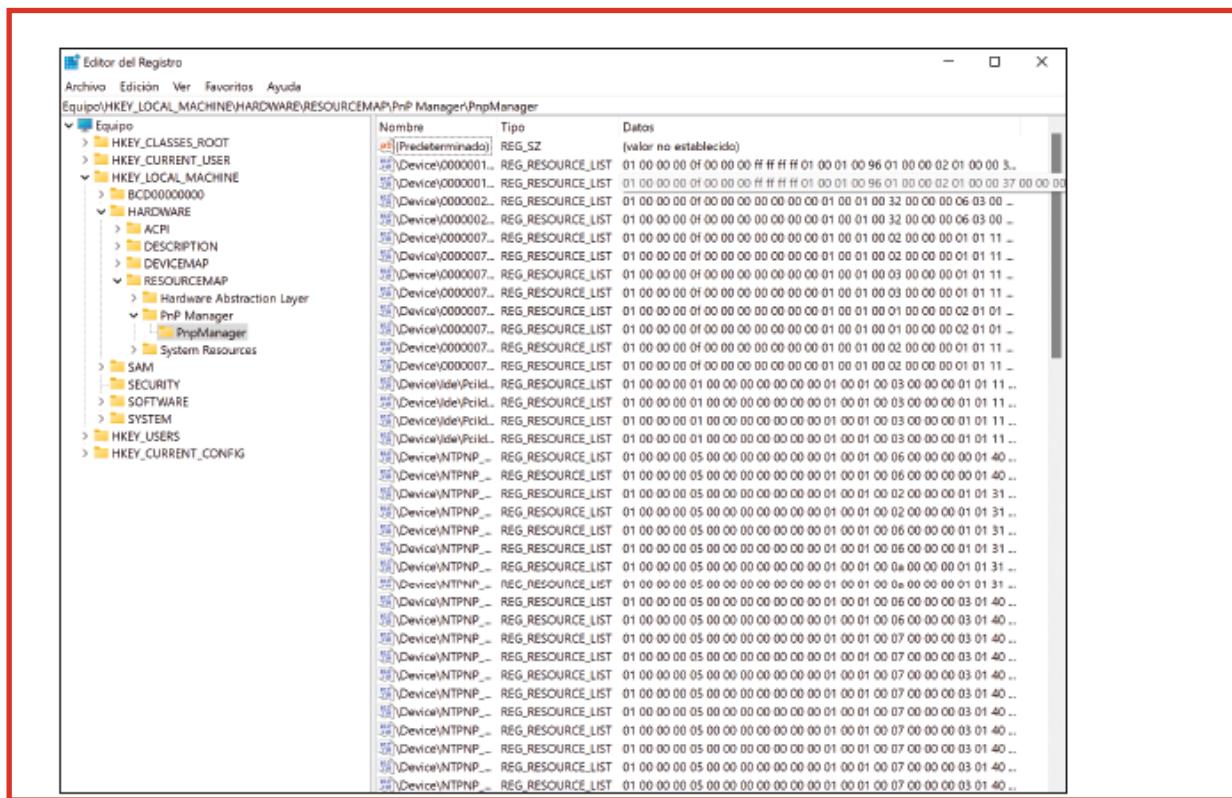
Lo que se pretende con esta tecnología es lo siguiente:

- Realizar la detección automática del *hardware* que se requiere instalar o desinstalar.
- Aislar al usuario de la configuración *hardware*, es decir, todo debe configurarse mediante una interfaz *software*.
- Cuando se conecte un dispositivo nuevo, los controladores de este deben instalarse de forma automática.
- El dispositivo deberá soportar que se pueda conectar “en caliente”, es decir, con todo el sistema en funcionamiento. Siempre que el bus de conexión lo permita.

### Instalación de dispositivos en Windows

Los sistemas *Windows* implementan la tecnología *Plug and Play* con varios componentes *software*:

- **Gestor *Plug and Play*:** este gestor trabaja en dos modos, usuario y *kernel*. El modo *kernel* es el que está en contacto directo con el *hardware* y el resto de componentes del núcleo, para gestionar la detección y la configuración del *hardware*.  
El modo usuario ofrece una interfaz que permite consultar y modificar la configuración del *software PnP* que se está instalando.
- **Gestor de energía:** este gestor se encarga de administrar la energía que le llega a los dispositivos. Puede cortar la energía que le llega a un dispositivo, de forma temporal, si este no está en uso, siempre teniendo en cuenta la naturaleza del dispositivo.
- **Registro de *Windows*:** en el Registro de *Windows* (*Regedit*) se almacena información sobre el *software* y el *hardware* instalados para dispositivos PnP. Permite a los controladores y a los componentes que utilizan estos dispositivos una forma para poder identificar y localizar qué recursos necesita un determinado dispositivo.



Captura de pantalla que muestra el registro de Windows con la información para dispositivos Pnp

- **Ficheros INF (ficheros de información de instalación):** para la instalación del controlador de un dispositivo es necesario que este dispositivo sea descrito de forma detallada a través de un fichero ".inf".
  - **Controladores Plug and Play.** Pueden ser de dos tipos:
    - WDM: son totalmente compatibles con PnP.
    - NT PnP: emplean características de la arquitectura PnP, pero no todas ellas (por ejemplo no soportan los IRP de PnP).



## Actividades

31. Busque información sobre la rutina AddDevice, los parámetros que recibe y la salida que se obtiene al utilizarla. ¿Son los mismos que en DriverEntry?

### ***Pautas que debe seguir un driver para soportar PnP***

Las pautas que debe seguir un *driver* para soportar PnP son:

1. **Debe contener una rutina DispatchPnP:** el administrador PnP va a enviar estas peticiones cuando se enumeran, reestructuran, así como cuando ocurra cualquier actividad PnP en el sistema.
2. **No debe buscar el hardware:** el administrador PnP es el responsable de determinar la presencia de dispositivos *hardware*. Cuando detecta un dispositivo, es notificado al *driver*, a través de la rutina AddDevice.
3. **No debe asignar recursos hardware:** el administrador PnP es el responsable de asignar los recursos a cada dispositivo y de notificar al *driver* cada asignación.

### Instalación de dispositivos en sistemas Unix

En *Unix/Linux* el programa de configuración de PnP encuentra todos los dispositivos PnP que están conectados al sistema y les pregunta qué recursos de bus necesitan, es decir, IRQ, DMA, etc.

Los dispositivos que no son PnP también tendrán recursos de bus reservados con los que no se puede contar.

En definitiva, la tarea que tiene que realizar PnP es asociar los dispositivos físicos con el *software*, es decir, con los *drivers*.

Para ello, PnP distribuye los siguientes **recursos de bus**: direcciones de E/S, regiones de memoria, IRQ, canales DMA.

PnP mantiene un registro con lo que se ha asignado y permite a los *drivers* obtener esta información.

Una vez que estos recursos hayan sido asignados, el *driver* actual y el fichero asociado al dispositivo que se encuentra en el directorio "/dev" están listos para ser usados.

PnP es un proceso realizado conjuntamente por *software* y *hardware*. Con *Linux* cada *driver* tiene su propio PnP y usa un *software* proporcionado por el *kernel*.

El *hardware* de la BIOS del PC hace PnP cuando se reinicia.

## 9. Herramientas

En este apartado se van a describir las herramientas de las que se va a hacer uso a la hora de desarrollar, depurar y verificar el controlador.

Tanto en sistemas *Windows* como *Linux*, antes de comenzar directamente con el desarrollo de los controladores, es necesario adecuar el entorno que se va a utilizar, en qué lugar del sistema va a estar almacenado el controlador, etc.

Una vez creado el entorno y desarrollado el controlador, se van a utilizar unas herramientas para depurar el código y realizar las pruebas pertinentes, de manera que permitan verificar las rutinas que componen el controlador y su funcionamiento dentro del sistema.

## 9.1. Entornos de desarrollo de controladores de dispositivo en sistemas operativos de uso común

Dentro de los sistemas de uso común se encuentran los que se describen a continuación.

### Sistemas Windows

Casi la totalidad de los entornos de desarrollo están preparados para trabajar en equipos con entorno *Microsoft Windows*, puesto que es el sistema operativo más extendido.

#### Preparar el entorno

##### Instalar Windows Driver Kit

Para el desarrollo de *drivers* en *Windows* se necesita **Windows Driver Kit (WDK)**, herramienta que permite la construcción, las pruebas y el desarrollo de *drivers*.

Al instalar WDK, también se incluye:

- La documentación necesaria para el desarrollo de controladores.
- Los ficheros que definen los tipos y cabeceras que constituyen la API de desarrollo de controladores (ficheros de extensión ".h").

Existen dos versiones de WDK dependiendo de la versión del sistema operativo al que se va a incorporar el nuevo *driver*. Las versiones son:

- **Windows Driver Kit 8.1:** esta herramienta se utiliza para desarrollar controladores para versiones anteriores a *Windows 8.1*.  
Está integrado en el IDE *Visual Studio* y proporciona un entorno para desarrollar, construir, empaquetar, probar y depurar controladores.
- **Windows Driver Kit 7.1.0:** se utiliza para desarrollar controladores para *Windows 7*, *Windows Vista* y *Windows XP*.  
No está integrado con *Visual Studio* y, por lo tanto, es necesario utilizar otras herramientas que lo complementen para escribir, probar y depurar el controlador.

## **Edición de los ficheros**

Para editar los ficheros se puede utilizar cualquier editor de texto plano, por ejemplo uno como *NotePad*, aunque es más recomendable utilizar alguno que resalte el código escrito en C/C++.

En la tabla que aparece a continuación pueden verse ejemplos de potentes editores de texto de *software libre*:

Plataforma	Editor	Comentarios
<b>Multiplataforma</b>	<i>Sublime Text</i>	Es muy popular y tiene un gran rendimiento.
	<i>NetBeans</i>	IDE que soporta múltiples lenguajes.
	<i>Eclipse</i>	IDE basado en Java, es adecuado para grandes proyectos.
<b>Windows</b>	<i>NotePad++</i>	Sustituye al editor que se instala por defecto.
<b>macOS</b>	<i>ATOM</i>	Personalizable y sencillo de utilizar
<b>Unix</b>	<i>Bluefish Editor</i>	Potente herramienta para desarrollar aplicaciones



## **Definición**

### **IDE**

Estas siglas corresponden a *Integrated Development Environment*, es decir, entorno integrado de desarrollo. Este entorno es un conjunto de herramientas útiles para programar. Normalmente ofrece un entorno para codificar, un compilador y un depurador de código (*debugger*) y un constructor de interfaz gráfica (GUI).

### **GUI**

Estas siglas corresponden a *Graphic User Interface*, es decir, Interfaz Gráfica de Usuario. Es un conjunto de formas y métodos que tienen como finalidad la interacción de un sistema con los usuarios, de la manera más cómoda, usando formas gráficas como botones, iconos, ventanas, etc., e imágenes.



## **Actividades**

32. Busque información sobre los distintos tipos de licencias de *software libre*: licencia MIT, *open source*, licencia GPL.
- 

## Sistemas Unix

Las herramientas que se necesitan en *Unix/Linux* son las que se describen a continuación.

Para el desarrollo de un *driver* se necesita un editor de texto, tal como se describía en el apartado anterior, o un entorno de desarrollo para C++.

Desde la consola es posible realizar el resto de las operaciones, compilarlo, depurarlo, etc., tal como se ha visto anteriormente. Pero, ya que para programar el *kernel* es necesario compilarlo también, se va a ver qué pasos hay que dar para compilar el *kernel* de *Linux*.

### **Obtener el código fuente del kernel**

El código fuente del *kernel* más actual está siempre disponible de dos formas: en un ***tarball*** o en parches incrementales que pueden descargarse desde el sitio oficial de *Linux Kernel* ([www.kernel.org](http://www.kernel.org)).



#### Nota

*Tarball* es un formato de archivo creado con el comando tar, que permite almacenar archivos y directorios en un solo archivo.

### **Instalar el código fuente del kernel**

El *tarball* del *kernel* es distribuido en estos formatos: GNU zip (gzip) y bzip2. Este último es el preferido porque generalmente comprime mejor que gzip.

Después de bajarse el código fuente, la descompresión y el *untarring* son muy sencillos.

```
$ tar xvjf linux-x.y.z.tar.bz2
```

Si está comprimido con bzip2.

```
$ tar xvzf linux-x.y.z.tar.gz
```

Si está comprimido con GNU gzip.

Esto descomprime y extrae el código fuente al directorio linux -x.y.z.

### **Usando parches**

Cuando se introducen cambios en el código fuente, se hace a través de parches. Los parches incrementales son una forma fácil de modificar el *kernel* sin tener que instalar el *kernel* entero.

Los parches se ejecutan de esta manera:

```
$ patch -p1 < ../patch-x.y.z
```

### **Configurando el kernel**

Antes de construir el *kernel* es necesario configurarlo. La herramienta más sencilla para ello es una utilidad en línea de comandos que se ejecuta así:

```
$ make config
```

Esta utilidad pasa por todas las opciones e interactúa con el usuario para seleccionar si interesa o no. Como lleva mucho tiempo, una opción más rápida es usar la versión gráfica.

```
$ make menuconfig o $ make gconfig
```

Una vez realizada la configuración, se puede construir/compilar el *kernel* con un solo comando:

```
$ make
```

### **Instalar un nuevo kernel**

Una vez construido, es necesario instalar el *kernel*. Para ello, se ejecuta este comando como *root*:

```
%make module_install
```

Esto instala todos los módulos compilados en sus directorios *home* bajo /lib/modules.

## **9.2. Herramientas de depuración y verificación de controladores de dispositivos**

Las herramientas de depuración van a permitir el control de los errores que puedan producirse en el desarrollo del nuevo *driver*.

Una vez que se hayan controlado los errores, se pasa a verificar el controlador para evitar que se produzcan fallos a la hora de integrarlo en el sistema.

Para ver las herramientas que se utilizan, se va a distinguir entre los dos sistemas operativos más usados.

### **Windows**

En Windows se utilizan principalmente las siguientes herramientas:

#### **Windows Driver Kit (WDK)**

En Windows, desde WDK es posible crear un entorno de pruebas y depuración. Para ello, se utilizan dos equipos, uno que sirve de *host* y el otro que corresponde al destino, también llamado equipo de prueba.

En el equipo *host* se desarrolla y compila el *driver*. Aquí es también donde se depura. Una vez que se realizan estas operaciones, se procede a la ejecución del *driver* en el otro equipo, el equipo destino.

El equipo *host* y el equipo destino tienen que estar en el mismo **dominio de red** o el mismo **grupo de trabajo** (si están en un grupo de trabajo se recomienda que se conecten a través de un *router* en lugar de un *hub* o un *switch*).



**Definición**

### **Dominio de red**

Conjunto de ordenadores que se encuentran conectados a una red, de manera que uno de estos equipos va a administrar los usuarios y los privilegios que tienen estos. Un dominio puede contener miles de equipos. Además, pueden encontrarse en distintas redes locales.

### **Grupo de trabajo**

Conjunto de ordenadores conectados a una red, pero todos están al mismo nivel. Normalmente no hay más de veinte equipos.

### **DebugView**

Esta aplicación permite la **monitorización** de los errores, en un sistema local o en cualquier ordenador que esté conectado a una red (que sea posible el acceso a través de TPC/IP).

Muestra los errores del modo *kernel* y del modo usuario y permite filtrar el modo en el que proceden los mensajes.

Además, permite ver el resultado de las llamadas que se hacen a las API, por ejemplo a la que se utiliza para depurar, **DbgPrint** (es parecida a la printf, pero para controladores).

La instalación y uso de esta herramienta es sencilla, solo ejecutando el fichero del programa “**dbgview.exe**” y comienza a capturar mensajes.

En la imagen que se muestra a continuación se pueden ver los mensajes que capturan desde un Win32, desde un sistema remoto. Cabe destacar que aparecen líneas resaltadas según el filtro que se haya utilizado.

```

DebugView on \\WINDOWS_11 (local)
File Edit Capture Options Computer Help
# Time Debug Print
123 0.12802450 [12296] en Microsoft.VisualStudio.Settings.Internal.StorageFactory`1.CreateInstance(String pa...
124 0.12802450 [12296] en Microsoft.VisualStudio.Settings.Internal.StorageFactory`1.<OpenAsync>d__10.MoveNext...
125 0.12802450 [12296] --- Fin del seguimiento de la pila de la ubicación anterior donde se produjo la excepción...
126 0.12802450 [12296] en System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
127 0.12802450 [12296] en System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotificatio...
128 0.12802450 [12296] en Microsoft.Developer.Settings.PublishingSettingStorageFactory.<DefaultStorageFactor...
129 0.12802450 [12296] --- Fin del seguimiento de la pila de la ubicación anterior donde se produjo la excepción...
130 0.12802450 [12296] en System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
131 0.12802450 [12296] en System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotificatio...
132 0.12802450 [12296] en Microsoft.Developer.Settings.SettingStorage.<OpenStorageAsync>d__20.MoveNext()
133 0.12802450 [12296] --- Fin del seguimiento de la pila de la ubicación anterior donde se produjo la excepción...
134 0.12802450 [12296] en System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
135 0.12802450 [12296] en System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotificatio...
136 0.12802450 [12296] en Microsoft.Developer.Settings.SettingStorage.<GetAsync>d__25.MoveNext()
137 0.12802450 [12296] --- Fin del seguimiento de la pila de la ubicación anterior donde se produjo la excepción...
138 0.12802450 [12296] en System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
139 0.12802450 [12296] en System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotificatio...
140 0.12802450 [12296] en Microsoft.Developer.Settings.SettingStorage.<GetIsRoamingEnabledAsync>d__44.MoveNext()
141 0.12802450 [12296] --- Fin del seguimiento de la pila de la ubicación anterior donde se produjo la excepción...
142 0.12802450 [12296] en System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
143 0.12802450 [12296] en System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotificatio...
144 0.12802450 [12296] en Microsoft.Developer.Settings.SettingsService.<PutSettingAsync>d__19.MoveNext()
145 0.12802450 [12296] --- (Nº de excepción interna 0) System.ObjectDisposedException: No se puede obtener acc...
146 0.12802450 [12296] Nombre del objeto: 'RefCountedCrossProcessLockFactory'.
147 0.12802450 [12296] en Microsoft.VisualStudio.PlatformUI.DisposableObject.ThrowIfDisposed()
148 0.12802450 [12296] en Microsoft.Developer.Settings.RefcountedCrossProcessLockFactory.<AcquireLockAsync>d...
149 0.12802450 [12296] <---
150 0.12802450 [12296] , PutSettingAsync, 122, Newtonsoft.Json.Linq.JToken[]
151 0.16718100 [12296] JsonRpc Error: 10 :
152 48.69881058 [1444] No storage handlers could store the final output file, skipping rundown actions

```

Captura de pantalla que muestra la herramienta DebugView

## Linux

Para la depurar el código en *Linux*, lo más utilizado es **printf**, como se ha explicado anteriormente.

Por lo tanto, la herramienta que se utiliza es la misma que para el desarrollo, ya que consiste en controlar los errores desde el código.

Otra forma de controlar los errores es habilitar todas las opciones de depuración del *kernel*. Esto se hace desde el código incluyendo esta sentencia:

```
#define assert(x) do { if (!(x) BUG(); } while(0)
```

Aunque se utilice la depuración por código, *Linux* cuenta con varios depuradores (*debuggers*). Los más usados son:

- **GDB.** Depurador GNU. Actualmente es el más usado y el más potente. Tiene soporte para una gran variedad de lenguajes de programación además de distintas arquitecturas de CPU.

- **LLDB.** Depurador con una gran velocidad que permite depurar código de *Xcode* para *macOS* o *Android Studio*.



## Actividades

33. Amplíe información sobre los dos *debuggers* que se han comentado para *Linux*.

## 10. Documentación de manejadores de dispositivos

La documentación en el *software* es una parte muy importante. En esta sección se va a explicar qué tipo de documentos debe incluir un controlador o *driver* bien especificado.

En la fase de desarrollo, la documentación se traduce en los comentarios que se introducen en el código. A través de ellos se comentan: variables, constantes, funciones y todo lo que sirva para esclarecer el código.

Esta información puede servir en caso de que el *driver* vaya a ser modificado en el futuro, por el desarrollador original o por otros que amplíen este código, o simplemente para hacer un seguimiento del código.

Este tipo de documentación está claramente enfocada a los desarrolladores.

Continuando con la documentación técnica, se va a exponer cómo son las especificaciones técnicas, que van dirigidas a un público experto.

Posteriormente, se va a explicar cómo elaborar dos manuales, uno para poder instalar el *driver* y otro para explicar cómo usarlo.

Ninguno de estos dos tiene por qué ir dirigido a un público experto, por lo que tienen que venir explicados en un lenguaje cómodo, sin abusar de los tecnicismos, que permita que el usuario, sea cual sea su formación técnica, consiga terminar instalando el *driver* y utilizando el dispositivo a través de la interfaz que ofrece este.

### **10.1. Elaboración de especificaciones técnicas siguiendo directrices específicas de sistemas operativos de uso común**

Las especificaciones técnicas son documentos que contienen normas y procedimientos que deben ser utilizados para la realización del controlador de dispositivo.

Es un documento con carácter técnico, de manera que recoge información del *hardware* del dispositivo y del *software* que lo controla.

Un ejemplo de lo que podría contener la especificación técnica para *Windows* es:

# Ficha técnica

## Sistemas operativos compatibles

Windows: 10, 8.1, 8, 7: 32 o 64 bits, 2 GB de espacio disponible en el disco duro, unidad de CD-ROM/DVD o conexión a Internet, puerto USB, Internet Explorer

Windows Vista: solo 32 bits, 2 GB de espacio disponible en el disco duro, unidad de CD-ROM/DVD o conexión a Internet, puerto USB, Internet Explorer 8

## Sistemas operativos compatibles

Apple OS X El Capitan (v10.11), Apple OS X Yosemite (v10.10), OS X Mavericks (v10.9), 1 GB de espacio disponible en el disco duro, se requiere Internet, USB

Linux (para obtener más información, visite <http://hplipopensource.com/hplip-web/index.html>)

Unix (para obtener más información, visite <http://www.hp.com/go/unixmodelscripts>)

Windows: 10, 8.1, 8, 7: 32 o 64 bits, 2 GB de espacio disponible en el disco duro, unidad de CD-ROM/DVD o conexión a Internet, puerto USB, Internet Explorer

Windows Vista: solo 32 bits, 2 GB de espacio disponible en el disco duro, unidad de CD-ROM/DVD o conexión a Internet, puerto USB, Internet Explorer 8

## Sistemas operativos de red compatibles

Apple OS X El Capitan (v10.11), Apple OS X Yosemite (v10.10), OS X Mavericks (v10.9), 1 GB de espacio disponible en el disco duro, se requiere Internet, USB

Linux (para obtener más información, visite <http://hplipopensource.com/hplip-web/index.html>)

Unix (para obtener más información, visite <http://www.hp.com/go/unixmodelscripts>)

## Sistema operativo (tipo de compatibilidad)

No es compatible con Windows XP (64 bits) ni con Windows Vista (64 bits). No todos los «SO compatibles» son compatibles con el software INBOX

Ejemplo de una ficha técnica de un driver

Cabe destacar en este ejemplo que aparece el certificado que tiene que pasar un driver para que sea distribuido en Windows (WHQL).

Para entornos Linux, desde la Linux Foundation se dan algunas pautas para las especificaciones técnicas. A continuación, se ve un ejemplo que describe la documentación necesaria para un sistema software de telecomunicación para Linux.

<b>Tipos de Requisitos/ Especificaciones</b>	<b>Descripción</b>
<b>Requisitos de información general</b>	Describe los requisitos y el formato de separación, la terminología utilizada en todo el documento, las repercusiones de las necesidades y los errores.
<b>Definición de requisitos de disponibilidad</b>	Describe usos y funcionalidad necesaria para los nodos disponibles y su recuperación.
<b>Definición de requisitos Clustering</b>	Describe los componentes útiles y necesarios para construir un conjunto agrupado de sistemas individuales. El objetivo clave es clustering de alta disponibilidad.
<b>Definición de requisitos de servicio</b>	Describe las características útiles y necesarias para los servicios y el mantenimiento de un sistema y las herramientas que soportan dicho servicio.
<b>Definición de requisitos de desempeño</b>	Describe las características útiles y necesarias que contribuyan a un adecuado rendimiento de un sistema.
<b>Definición de requerimientos de estándares</b>	Proporciona referencias a las API necesarias y especificaciones estándar.
<b>Definición de requisitos hardware</b>	Describe el uso y las necesidades hardware específicas.
<b>Definición de requisitos de seguridad</b>	Describe las características útiles y necesarias para la construcción de sistemas seguros.



### Actividades

34. Busque información de especificaciones técnicas para un *driver* de un dispositivo común en *Windows* y en *Linux*.
- 

## 10.2. Elaboración del manual de instalación

El manual de instalación es un documento que debe acompañar a todo tipo de *software*, en este caso al controlador del dispositivo.

Debe estar bien estructurado, además de explicar de forma clara los pasos a seguir para que un usuario que va a utilizar el nuevo *driver* pueda instalarlo con éxito en su sistema.

El manual de instalación va a constar de las siguientes partes:

## 1. Primera página

La primera página debe contener información general sobre:

- El nombre del controlador del dispositivo.
- El autor/a.
- La licencia de distribución, si es de distribución libre, si está sujeto a alguna restricción etc.

## 2. Tabla de contenidos

En este apartado debe aparecer un índice con la información que va contenida en el resto del documento. La información debe aparecer de una forma jerárquica, acompañada del número de página donde encontrar esta información, para que la búsqueda en el documento sea fácil e intuitiva.

## 3. Introducción

Esta parte introduce al usuario en la necesidad por la que se creó el documento. Consta de los siguientes apartados:

### 3.1. Objetivo

El objetivo del documento debe quedar claro y explicado con verbos en infinitivo.

En este caso, uno de los principales objetivos debe ser orientar al usuario en la instalación del *driver*.

### 3.2. Referencias

En este apartado deben aparecer lecturas que completen la información del documento. También pueden aparecer *links* a páginas web que amplíen la información y que se escapan de los objetivos del documento, pero que es importante tener en cuenta (por ejemplo: dar información sobre conocimientos o términos que van a aparecer en el documento y que se dan por supuestos).

## 4. Manual

Aquí es donde se van a dar las indicaciones concretas para la instalación del *driver*.

#### **4.1. Requisitos previos**

En este apartado se debe incluir información sobre los requerimientos mínimos de *hardware* o de *software* que se necesitan para poder instalar el *driver*.

Por ejemplo, un requerimiento puede ser: "Necesita ser instalado en un PC que corra bajo el sistema operativo *Windows*".

#### **4.2. Instalación**

En este apartado se exponen los pasos detallados que se necesitan para la instalación. Incluye qué herramientas hay que utilizar (por ejemplo, si el *driver* está comprimido, indica que se necesita el uso de determinado programa para descomprimirlo, etc.).

Este apartado dependerá de la naturaleza del *driver* en cuestión.



#### **Consejo**

Para que la información sea más fácil de seguir, conviene incluir imágenes de las pantallas por las que hay que pasar hasta obtener éxito en la instalación.

#### **4.3. Configuración**

Si para el correcto funcionamiento del *driver* y como consecuencia del dispositivo es necesario configurar parámetros, registrar claves, etc., es necesario incluirlo en este apartado de forma clara.

#### **5. Anexos**

En este apartado se incluirá información útil para la instalación, pero que se escapa de lo que aparece en el manual, aunque tiene que tener relación con el contenido.

### **10.3. Elaboración de manual de uso**

El manual de uso es un documento que va dirigido a todo tipo de usuarios, desde los que tienen conocimientos informáticos más profundos hasta aquellos que solo quieren hacer funcionar un dispositivo en su equipo.

Por lo tanto, este manual debe ser muy sencillo y conciso, constando de:

## **1. Primera página**

La primera página debe contener información general:

- El nombre del controlador del dispositivo.
- El autor/a y forma de contacto.
- La licencia de distribución, si es de distribución libre, si está sujeto a alguna restricción, etc.
- Versión del manual.

## **2. Tabla de contenidos**

Al igual que se indicaba en el manual de instalación, aquí va a aparecer un índice con la información que contiene el manual acompañado del número de página para que la localización sea fácil.

## **3. Definición del producto**

El objetivo de este apartado es explicar qué es el *driver* que tiene instalado y para qué se puede utilizar. Debe ser un texto que invite al usuario a utilizarlo.

## **4. Forma de acceso**

En esta parte se explica cómo puede acceder a la interfaz que está asociada al *driver*.

Debe mostrar dónde encontrar este *software* dentro del sistema del usuario.

## **5. Interfaz**

Aquí se describe la interfaz con la que el usuario interacciona con el dispositivo a través de su *driver*.

Debe aparecer detallado qué es lo que va a encontrar en la interfaz y cómo utilizar cada una de sus opciones, pantallas y configuraciones.

## **6. Anexos**

Este apartado, al igual que en el caso del manual de instalación, ampliará la información que está en conexión con la del manual, pero que se escapa del contenido del mismo.



### **Actividades**

35. Busque un manual de usuario de un *driver* de uso común para *Windows* y para *Linux*.
- 

## 11. Resumen

A lo largo de este capítulo, se ha ofrecido una explicación detallada de qué es un controlador de dispositivo.

Desde cómo se relaciona con el sistema operativo que lo contiene, hasta en qué modo opera: en modo usuario o en modo *kernel* o núcleo.

Se han mostrado los distintos tipos de controladores que hay y las diferencias principales, describiendo estos tres tipos: bloque, carácter y paquete.

Para empezar a programar un controlador o *driver* es necesario tener conocimientos de cómo funciona el sistema operativo, qué hacen las interrupciones, cómo se hacen transferencias de datos con DMA, para qué se utilizan los búferes, puertos, etc. Además, es necesario tener un conocimiento profundo del dispositivo.

Se ha visto el entorno para programar *drivers*, ya sea bajo *Windows* o bajo *Unix/Linux*, con las características y las herramientas que se utilizan en cada uno. Cabe destacar que es muy probable que a la hora de programas driver aparezcan errores, por ello se han descrito técnicas y mecanismos para detectarlos y para poder gestionarlos (mediante trazas, monitorizándolos, con *debuggers*, etc.).

Ya de lleno en la programación se han definido las diferencias a la hora de implementar el *driver* en una plataforma *Windows* y en una *Linux*. Teniendo en cuenta que hay que cumplir unos estándares de calidad para que el producto sea eficiente.

Una vez desarrollado el *driver* se han utilizado las herramientas de compilación y carga del controlador, en las distintas plataformas. Además, también se han dado las pautas para poder incluirlo en las nuevas versiones de los sistemas operativos, es decir, para que pueda ser distribuido. Por último, se ha aprendido lo importante que es la documentación del *driver*, qué tipos de documentos deben acompañarlo y a quién van dirigidos.

# Bibliografía

## Monografías

- | ALEGRE Ramos, M. P: *Sistemas operativos monopuesto*. Madrid: Editorial Paraninfo, 2019.
- | ACERA García, M. A.: *C/C++*. *Curso de programación*. Madrid: Editorial Anaya Multimedia, 2017.
- | BILLIMORIA, K. N.: *Linux Kernel Debugging: Leverage proven tools and advanced techniques to effectively debug Linux kernels and kernel modules*. Birmingham: Editorial Packt Publishing, 2022.
- | MORENO Muñoz, A. y CÓRCOLES Córcoles, S.: *Aprende Java en un fin de semana*. Editorial Independently published, 2021.
- | PONS, N.: *Linux. Domine los comandos básicos*. Barcelona: Editorial ENI, 2022.
- | ORBET, J., RUBINI.A, KROAH-HARTMAN, G.: *Linux Device Drivers*. Editorial O'Reilly Media, 2005.

## Textos electrónicos, bases de datos y programas informáticos

- | Microsoft Developer, de: <<https://developer.microsoft.com/es-es/>>.
- | Linux, de: <<https://www.linux.org>>.
- | Universidad Complutense de Madrid, Facultad de Informática, Estructura de Computadores, de: <<http://www.fdi.ucm.es/profesor/jjruz/web2/>>.
- | Linux Professional Institute, de: <<https://www.lpi.org/es>>
- | Xavier Calbet (GULIC, Breve tutorial para escribir *drivers* en *Linux*), de: <[https://users.exa.unicen.edu.ar/catedras/rtlinux/material/apuntes/driv\\_tut\\_last.pdf](https://users.exa.unicen.edu.ar/catedras/rtlinux/material/apuntes/driv_tut_last.pdf)>.
- | Linux Foundation, de: <<https://www.linuxfoundation.org>>.
- | Linux Básico, de: <<https://linuxbasico.com>>.